Iris

Release 3.1.0

Iris Developers

Sep 17, 2021

GETTING STARTED

1 Installing Iris 3

2 Gallery 7

3 Introduction 79

4 Iris Data Structures 81

5 Loading Iris Cubes 87

6 Saving Iris Cubes 95

7 Navigating a Cube 99

8 Subsetting a Cube 105

9 Real and Lazy Data 111

10 Plotting a Cube 115

11 Cube Interpolation and Regridding 131

12 Merge and Concatenate 141

13 Cube Statistics 151

14 Cube Maths 157

15 Citing Iris 161

16 Code Maintenance 163

17 Introduction 165

18 Metadata 167

19 Lenient Metadata 183

20 Lenient Cube Maths 191

21 Getting Involved 197

22 Working With Iris Source Code 199

23 Contributing to the Documentation 209

24 Contributing to the Code Base 213

25 Contributing Your Changes 231

26 Releases 235

27 Iris API 239

28 What's New in Iris 467

29 Iris Technical Papers 531

30 Iris Copyright, Licensing and Contributors 539

Bibliography 541

Python Module Index 543

Index 545

A powerful, format-agnostic, community-driven Python package for analysing and visualising Earth science data.

Iris implements a data model based on the CF conventions giving you a powerful, format-agnostic interface for working with your data. It excels when working with multi-dimensional Earth Science data, where tabular representations become unwieldy and inefficient.

CF Standard names, units, and coordinate metadata are built into Iris, giving you a rich and expressive interface for maintaining an accurate representation of your data. Its treatment of data and associated metadata as first-class objects includes:

- visualisation interface based on matplotlib and cartopy,
- unit conversion,
- subsetting and extraction,
- merge and concatenate,
- aggregations and reductions (including min, max, mean and weighted averages),
- interpolation and regridding (including nearest-neighbor, linear and area-weighted), and
- operator overloads (+, -, *, /, etc.).

A number of file formats are recognised by Iris, including CF-compliant NetCDF, GRIB, and PP, and it has a plugin architecture to allow other formats to be added seamlessly.

Building upon NumPy and dask, Iris scales from efficient single-machine workflows right through to multi-core clusters and HPC. Interoperability with packages from the wider scientific Python ecosystem comes from Iris' use of standard NumPy/dask arrays as its underlying data storage.

Iris is part of SciTools, for more information see https://scitools.org.uk/. For **Iris 2.4** and earlier documentation please see the legacy documentation.

Install Iris as a user or developer.

Installing Iris

Example code to create a variety of plots.

Gallery

Find out what has recently changed in Iris.

What's New

Learn how to use Iris.

User Guide

Browse full Iris functionality by module.

Iris API

As a developer you can contribute to Iris.

Getting Involved

CHAPTER

ONE

INSTALLING IRIS

Iris is available using conda for the following platforms:

- Linux 64-bit,
- Mac OSX 64-bit, and
- Windows 64-bit.

Windows 10 now has support for Linux distributions via WSL (Windows Subsystem for Linux). This is a great option to get started with Iris for users and developers. Be aware that we do not currently test against any WSL distributions.

Note: Iris is currently supported and tested against Python 3.7 and 3.8.

Note: This documentation was built using Python 3.8.12.

1.1 Installing Using Conda (Users)

To install Iris using conda, you must first download and install conda, for example from https://docs.conda.io/en/latest/miniconda.html.

Once conda is installed, you can install Iris using conda with the following command:

```
conda install -c conda-forge iris
```

If you wish to run any of the code in the gallery you will also need the Iris sample data. This can also be installed using conda:

```
conda install -c conda-forge iris-sample-data
```

Further documentation on using conda and the features it provides can be found at https://docs.conda.io/projects/conda/en/latest/index.html.

1.2 Installing from Source Without Conda on Debian-Based Linux Distros (Developers)

Iris can also be installed without a conda environment. The instructions in this section are valid for Debian-based Linux distributions (Debian, Ubuntu, Kubuntu, etc.).

Iris and its dependencies need some shared libraries in order to work properly. These can be installed with apt:

```
sudo apt-get install python3-pip python3-tk libudunits2-dev libproj-dev proj-bin libgeos-
↪dev libcunit1-dev
```

The rest can be done with pip:

```
pip3 install scitools-iris
```

This procedure was tested on a Ubuntu 20.04 system on the 26th of July, 2021. Be aware that through updates of the involved Debian packages, dependency conflicts might arise or the procedure might have to be modified.

1.3 Installing from Source with Conda (Developers)

The latest Iris source release is available from https://github.com/SciTools/iris.

For instructions on how to obtain the Iris project source from GitHub see *Making Your own Copy (fork) of Iris* and *Set up Your Fork* for instructions.

Once conda is installed, you can install Iris using conda and then activate it. The example commands below assume you are in the root directory of your local copy of Iris:

```
conda env create --force --file=requirements/ci/iris.yml
conda activate iris-dev
```

The `--force` option is used when creating the environment, this is optional and will force the any existing `iris-dev` conda environment to be deleted first if present. This is useful when rebuilding your environment due to a change in requirements.

The `requirements/ci/iris.yml` file defines the Iris development conda environment *name* and all the relevant *top level conda-forge* package dependencies that you need to **code**, **test**, and **build** the documentation. If you wish to minimise the environment footprint, simply remove any unwanted packages from the requirements file e.g., if you don't intend to run the Iris tests locally or build the documentation, then remove all the packages from the *testing* and *documentation* sections.

Note: The `requirements/ci/iris.yml` file will always use the latest Iris tested Python version available. For all Python versions that are supported and tested against by Iris, view the contents of the requirements/ci directory.

Finally you need to run the command to configure your shell environment to find your local Iris code:

```
python setup.py develop
```

1.4 Running the Tests

To ensure your setup is configured correctly you can run the test suite using the command:

```
python setup.py test
```

For more information see *Running the Tests*.

1.5 Custom Site Configuration

The default site configuration values can be overridden by creating the file `iris/etc/site.cfg`. For example, the following snippet can be used to specify a non-standard location for your dot executable:

```
[System]
dot_path = /usr/bin/dot
```

An example configuration file is available in `iris/etc/site.cfg.template`. See *iris.config()* for further configuration options.

CHAPTER

TWO

GALLERY

The gallery is divided into sections as described below. All entries show the code used to produce the example plot. Additionally there are links to download the code directly as source or as part of a jupyter notebook, these links are at the bottom of the page.

In order to successfully view the jupyter notebook locally so you may experiment with the code you will need an environment setup with the appropriate dependencies, see *Installing Iris* for instructions. Ensure that `iris-sample-data` is installed as it is used in the gallery. Additionally ensure that you install `jupyter`. The command to install both is:

```
conda install -c conda-forge iris-sample-data jupyter
```

Once you have downloaded the notebooks (bottom of each gallery page), you may start the jupyter notebook via:

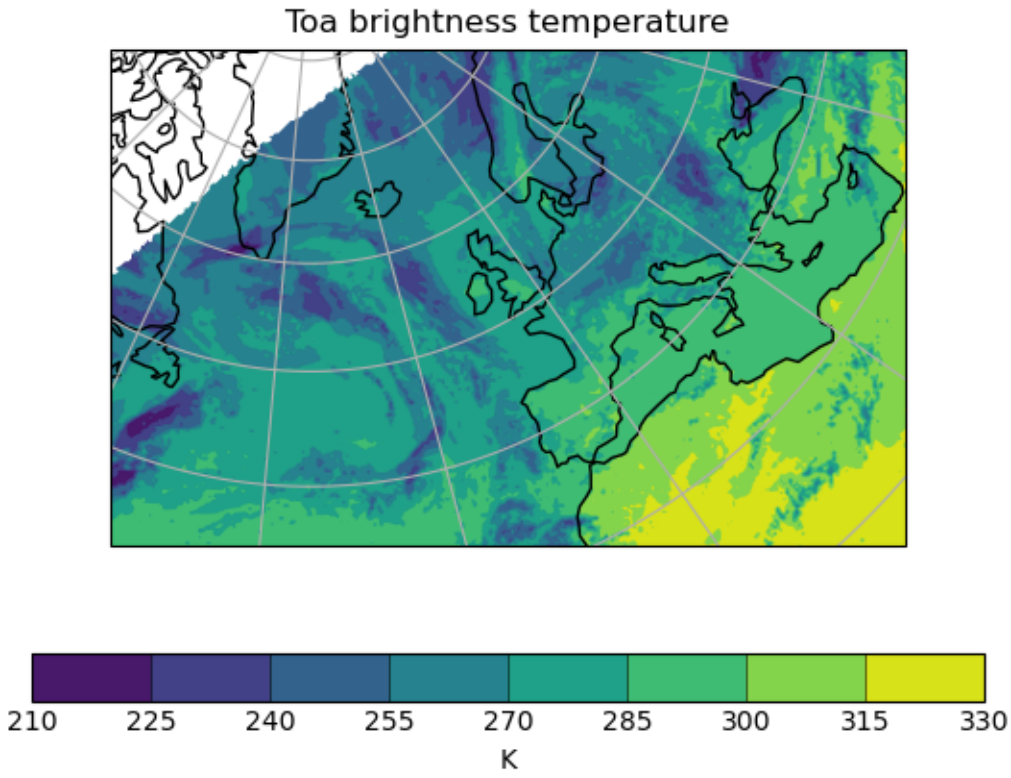
```
jupyter notebook
```

If you wish to contribute to the gallery see the *Gallery* section of the *Contributing to the Documentation*.

2.1 General

2.1.1 Example of a Polar Stereographic Plot

Demonstrates plotting data that are defined on a polar stereographic projection.

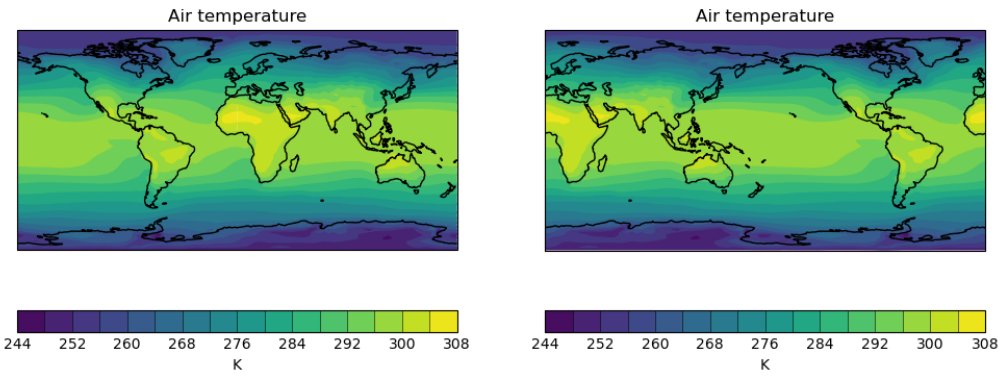



```
import matplotlib.pyplot as plt

import iris
import iris.plot as iplt
import iris.quickplot as qplt


def main():
    file_path = iris.sample_data_path("toa_brightness_stereographic.nc")
    cube = iris.load_cube(file_path)
    qplt.contourf(cube)
    ax = plt.gca()
    ax.coastlines()
    ax.gridlines()
    iplt.show()


if __name__ == "__main__":
    main()
```

Total running time of the script: (0 minutes 0.951 seconds)

2.1.2 Quickplot of a 2D Cube on a Map

This example demonstrates a contour plot of global air temperature. The plot title and the labels for the axes are automatically derived from the metadata.

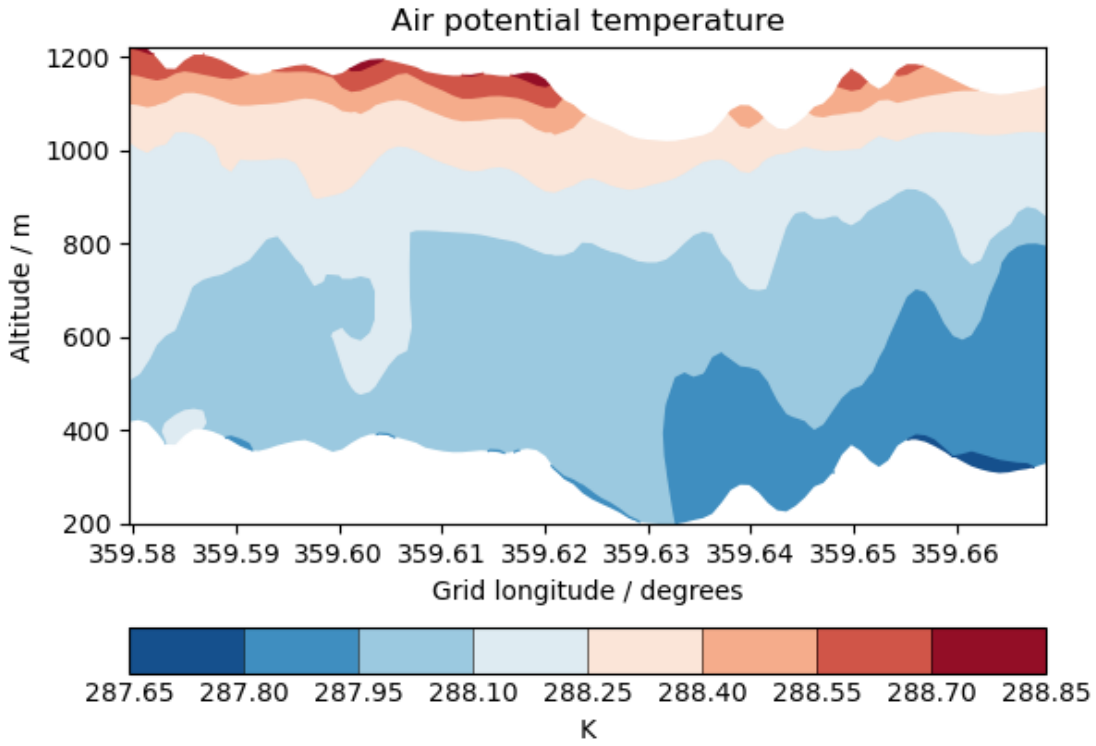



```
import cartopy.crs as ccrs
import matplotlib.pyplot as plt

import iris
import iris.plot as iplt
import iris.quickplot as qplt


def main():
    fname = iris.sample_data_path("air_temp.pp")
    temperature = iris.load_cube(fname)

    # Plot #1: contourf with axes longitude from -180 to 180
    plt.figure(figsize=(12, 5))
    plt.subplot(121)
    qplt.contourf(temperature, 15)
    plt.gca().coastlines()

    # Plot #2: contourf with axes longitude from 0 to 360
    proj = ccrs.PlateCarree(central_longitude=-180.0)
    plt.subplot(122, projection=proj)
    qplt.contourf(temperature, 15)
    plt.gca().coastlines()
    iplt.show()


if __name__ == "__main__":
    main()
```

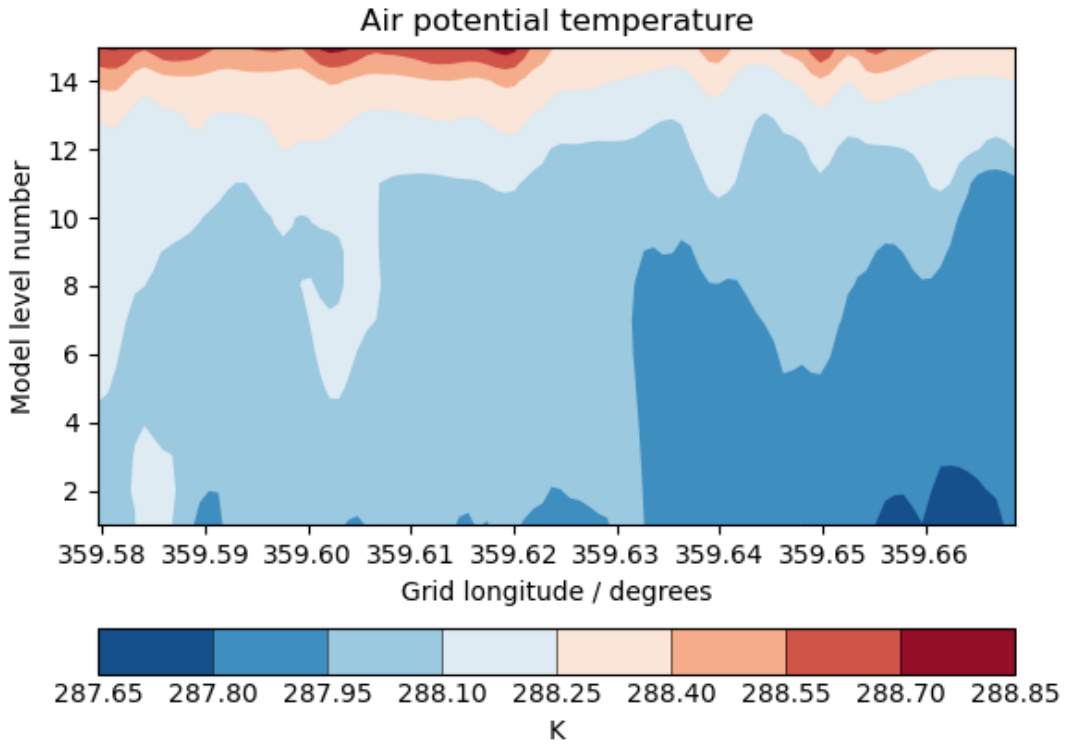
Total running time of the script: (0 minutes 0.785 seconds)

2.1.3 Cross Section Plots

This example demonstrates contour plots of a cross-sectioned multi-dimensional cube which features a hybrid height vertical coordinate system.



-



-

```
import matplotlib.pyplot as plt

import iris
import iris.plot as iplt
import iris.quickplot as qplt


def main():
    # Load some test data.
    fname = iris.sample_data_path("hybrid_height.nc")
    theta = iris.load_cube(fname, "air_potential_temperature")

    # Extract a single height vs longitude cross-section. N.B. This could
    # easily be changed to extract a specific slice, or even to loop over *all*
    # cross section slices.
    cross_section = next(
        theta.slices(["grid_longitude", "model_level_number"])
    )

    qplt.contourf(
        cross_section, coords=["grid_longitude", "altitude"], cmap="RdBu_r"
    )
    iplt.show()
```

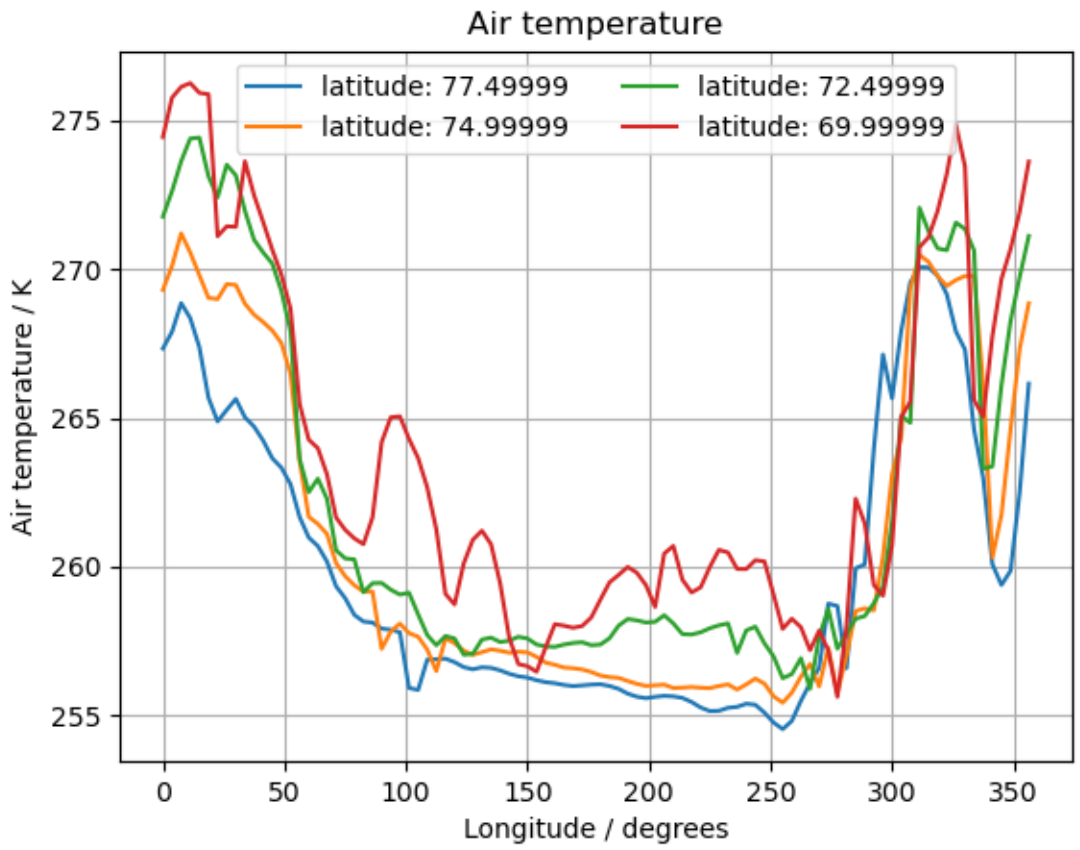
(continues on next page)

(continued from previous page)

```
    # Now do the equivalent plot, only against model level
    plt.figure()

    qplt.contourf(
        cross_section,
        coords=["grid_longitude", "model_level_number"],
        cmap="RdBu_r",
    )
    iplt.show()


if __name__ == "__main__":
    main()
```

Total running time of the script: (0 minutes 0.490 seconds)

2.1.4 Multi-Line Temperature Profile Plot




```
import matplotlib.pyplot as plt

import iris
```

(continues on next page)

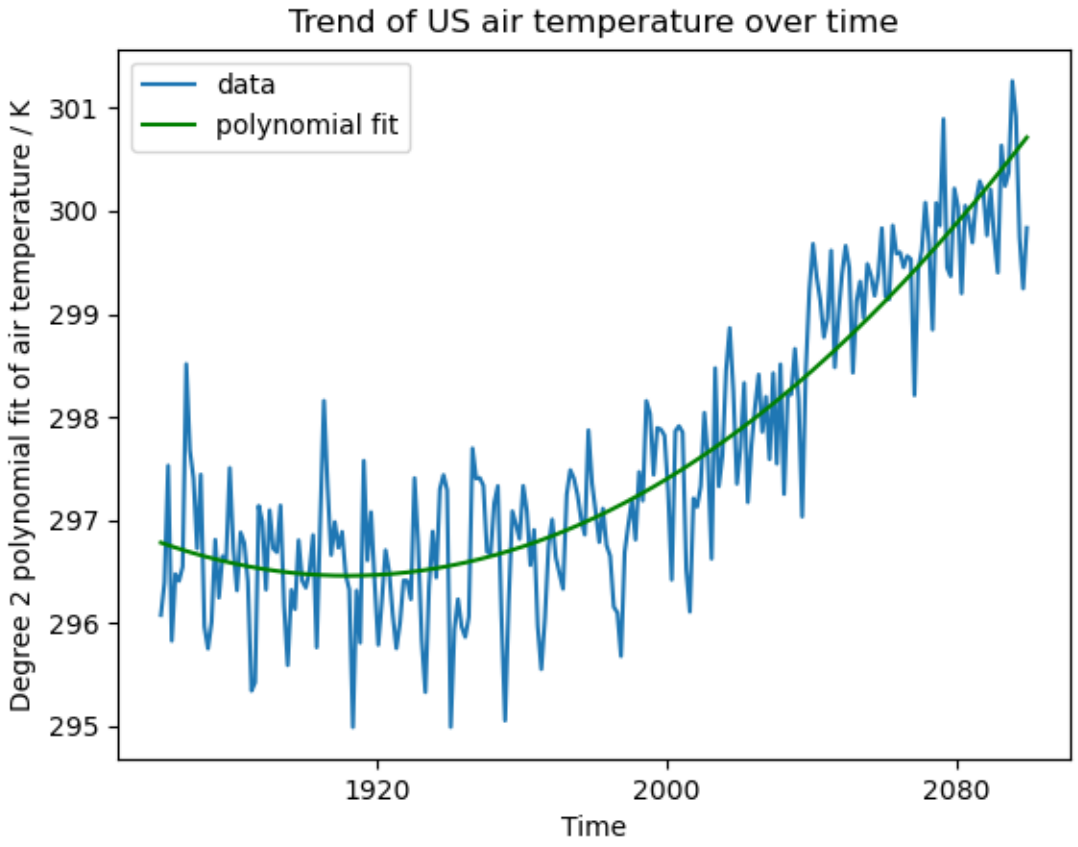
(continued from previous page)

```
import iris.plot as iplt
import iris.quickplot as qplt


def main():
    fname = iris.sample_data_path("air_temp.pp")

    # Load exactly one cube from the given file.
    temperature = iris.load_cube(fname)

    # We only want a small number of latitudes, so filter some out
    # using "extract".
    temperature = temperature.extract(
        iris.Constraint(latitude=lambda cell: 68 <= cell < 78)
    )

    for cube in temperature.slices("longitude"):

        # Create a string label to identify this cube (i.e. latitude: value).
        cube_label = "latitude: %s" % cube.coord("latitude").points[0]

        # Plot the cube, and associate it with a label.
        qplt.plot(cube, label=cube_label)

    # Add the legend with 2 columns.
    plt.legend(ncol=2)

    # Put a grid on the plot.
    plt.grid(True)

    # Tell matplotlib not to extend the plot axes range to nicely
    # rounded numbers.
    plt.axis("tight")

    # Finally, show it.
    iplt.show()


if __name__ == "__main__":
    main()
```

Total running time of the script: (0 minutes 0.176 seconds)

2.1.5 Fitting a Polynomial

This example demonstrates computing a polynomial fit to 1D data from an Iris cube, adding the fit to the cube's metadata, and plotting both the 1D data and the fit.




```
import matplotlib.pyplot as plt
import numpy as np

import iris
import iris.quickplot as qplt


def main():
    # Load some test data.
    fname = iris.sample_data_path("A1B_north_america.nc")
    cube = iris.load_cube(fname)

    # Extract a single time series at a latitude and longitude point.
    location = next(cube.slices(["time"]))

    # Calculate a polynomial fit to the data at this time series.
    x_points = location.coord("time").points
    y_points = location.data
    degree = 2
```

(continues on next page)

(continued from previous page)

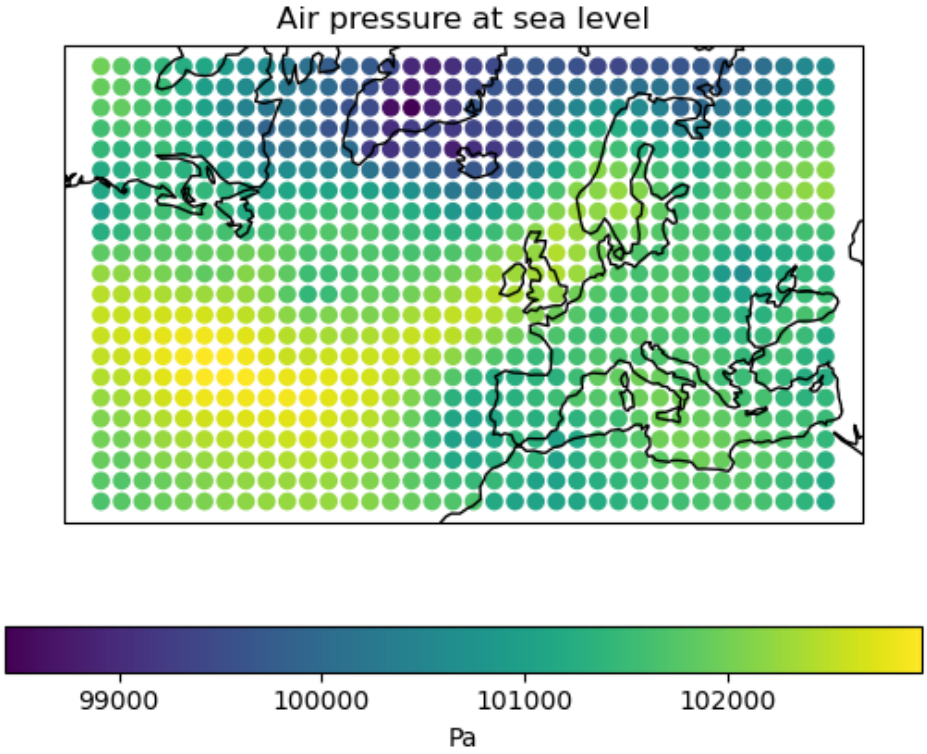
```
    p = np.polyfit(x_points, y_points, degree)
    y_fitted = np.polyval(p, x_points)

    # Add the polynomial fit values to the time series to take
    # full advantage of Iris plotting functionality.
    long_name = "degree_{}_polynomial_fit_of_{}".format(degree, cube.name())
    fit = iris.coords.AuxCoord(
        y_fitted, long_name=long_name, units=location.units
    )
    location.add_aux_coord(fit, 0)

    qplt.plot(location.coord("time"), location, label="data")
    qplt.plot(
        location.coord("time"),
        location.coord(long_name),
        "g-",
        label="polynomial fit",
    )
    plt.legend(loc="best")
    plt.title("Trend of US air temperature over time")

    qplt.show()


if __name__ == "__main__":
    main()
```

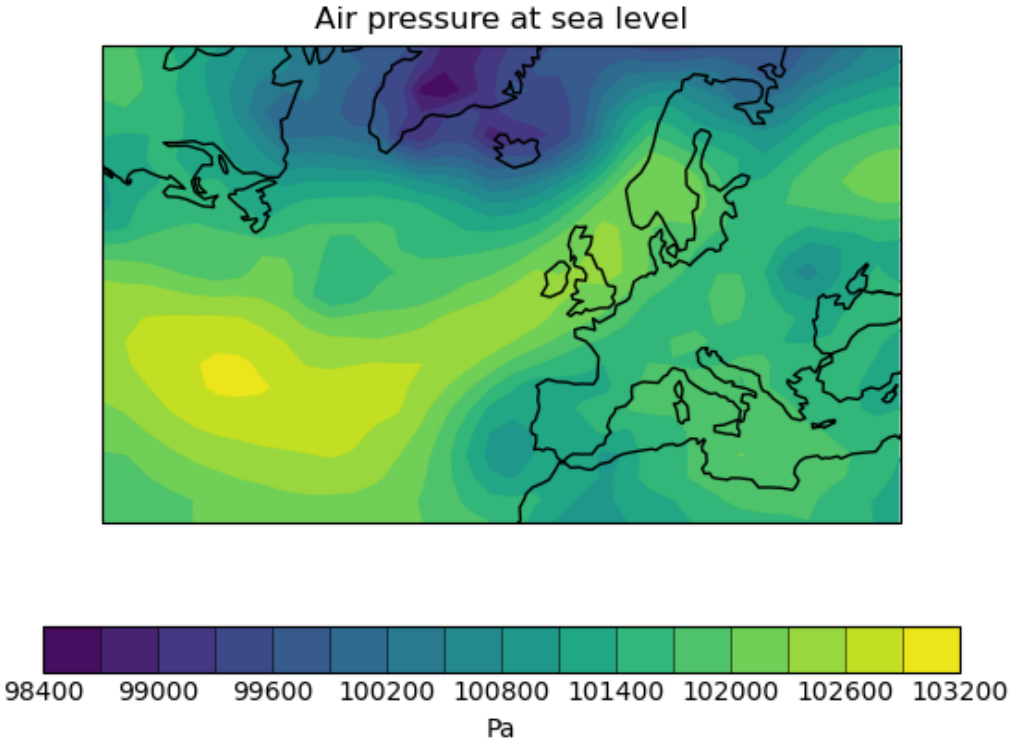
Total running time of the script: (0 minutes 0.243 seconds)

2.1.6 Rotated Pole Mapping

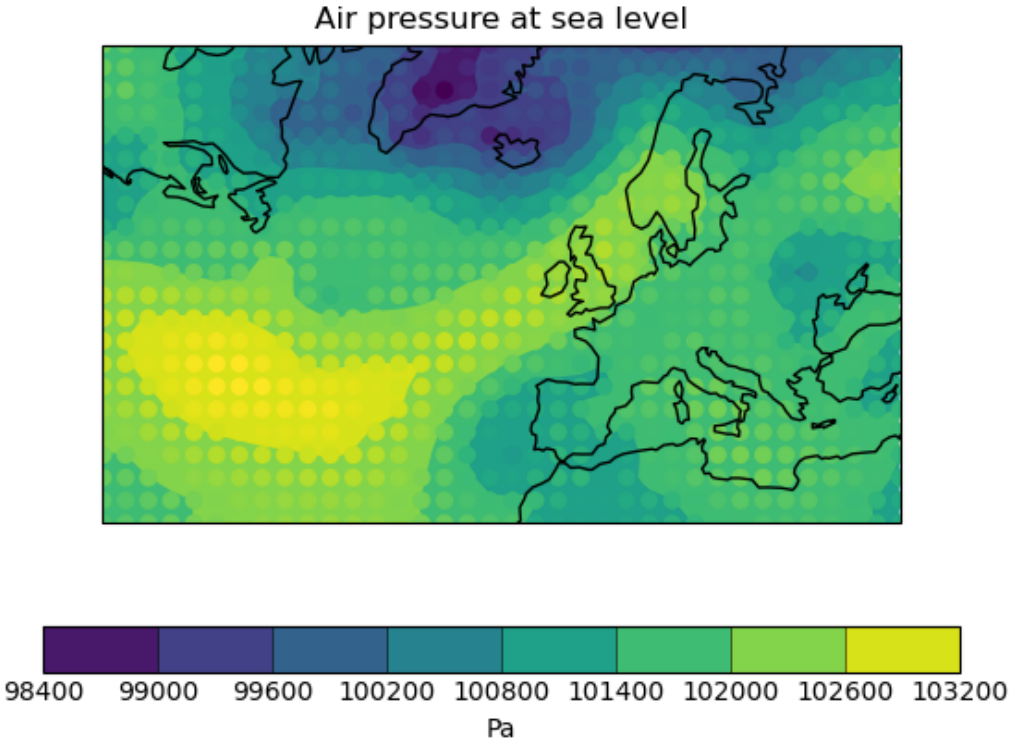
This example uses several visualisation methods to achieve an array of differing images, including:

- Visualisation of point based data
- Contouring of point based data
- Block plot of contiguous bounded data
- Non native projection and a Natural Earth shaded relief image underlay

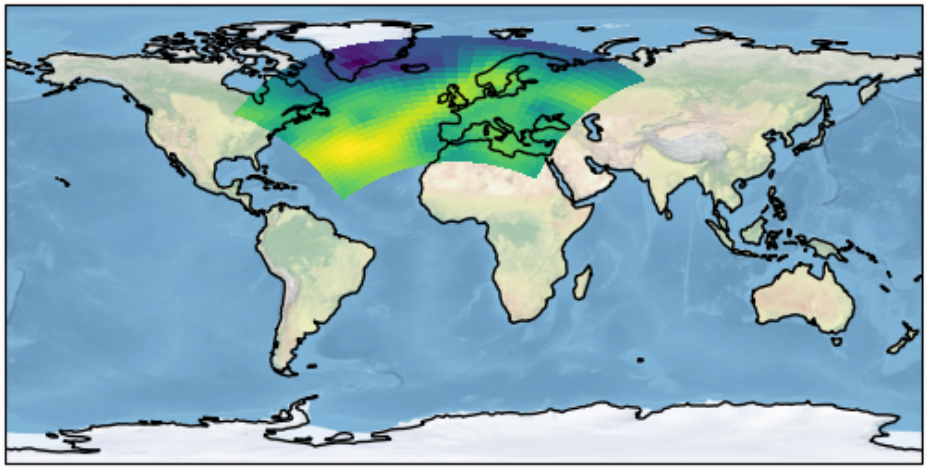

Air pressure at sea level
99000
100000
101000
102000
Pa


Air pressure at sea level
98400
99000
99600
100200
100800
101400
102000
102600
103200
Pa

-


Air pressure at sea level
98400
99000
99600
100200
100800
101400
102000
102600
103200
Pa

-



-

```
import cartopy.crs as ccrs
import matplotlib.pyplot as plt

import iris
import iris.analysis.cartography
import iris.plot as iplt
import iris.quickplot as qplt


def main():
    # Load some test data.
    fname = iris.sample_data_path("rotated_pole.nc")
    air_pressure = iris.load_cube(fname)

    # Plot #1: Point plot showing data values & a colorbar
    plt.figure()
    points = qplt.points(air_pressure, c=air_pressure.data)
    cb = plt.colorbar(points, orientation="horizontal")
    cb.set_label(air_pressure.units)
    plt.gca().coastlines()
    iplt.show()

    # Plot #2: Contourf of the point based data
    plt.figure()
```

(continues on next page)

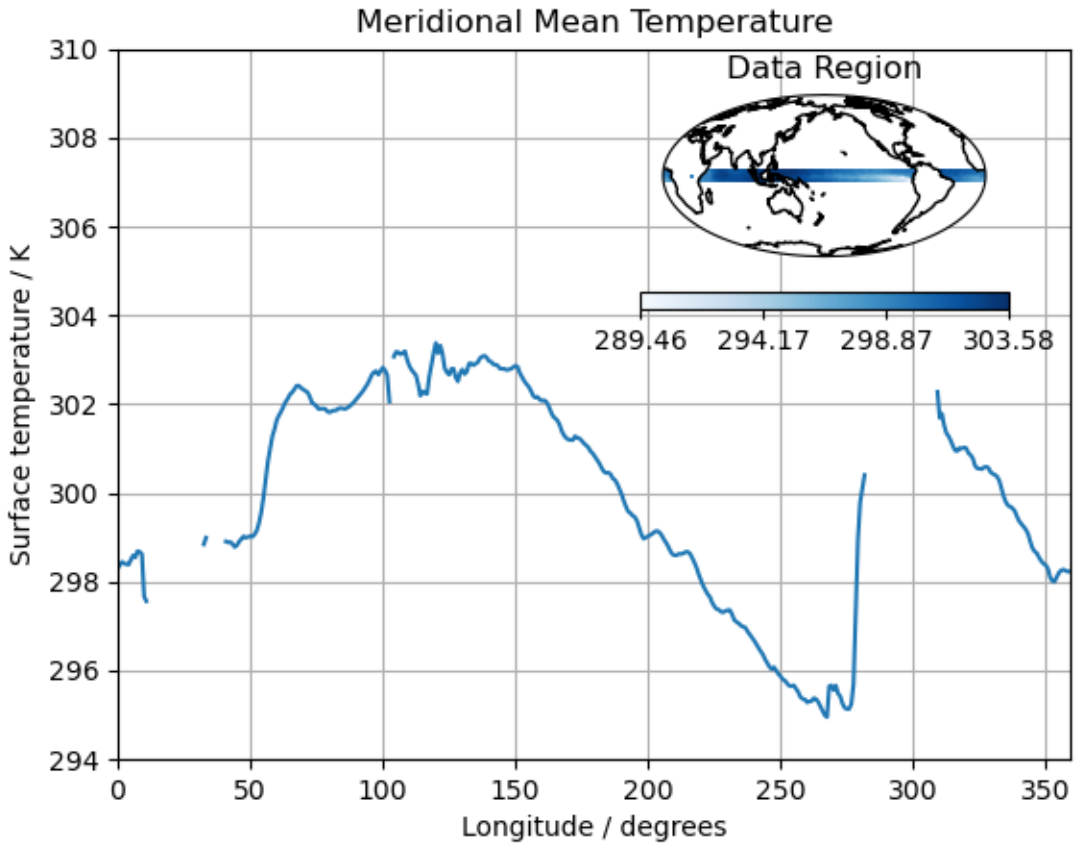
(continued from previous page)

```
    qplt.contourf(air_pressure, 15)
    plt.gca().coastlines()
    iplt.show()

    # Plot #3: Contourf overlayed by coloured point data
    plt.figure()
    qplt.contourf(air_pressure)
    iplt.points(air_pressure, c=air_pressure.data)
    plt.gca().coastlines()
    iplt.show()

    # For the purposes of this example, add some bounds to the latitude
    # and longitude
    air_pressure.coord("grid_latitude").guess_bounds()
    air_pressure.coord("grid_longitude").guess_bounds()

    # Plot #4: Block plot
    plt.figure()
    plt.axes(projection=ccrs.PlateCarree())
    iplt.pcolormesh(air_pressure)
    plt.gca().stock_img()
    plt.gca().coastlines()
    iplt.show()


if __name__ == "__main__":
    main()
```

Total running time of the script: (0 minutes 0.772 seconds)

2.1.7 Test Data Showing Inset Plots

This example demonstrates the use of a single 3D data cube with time, latitude and longitude dimensions to plot a temperature series for a single latitude coordinate, with an inset plot of the data region.




```
import cartopy.crs as ccrs
import matplotlib.pyplot as plt
import numpy as np

import iris
import iris.plot as iplt
import iris.quickplot as qplt


def main():
    cube1 = iris.load_cube(iris.sample_data_path("ostia_monthly.nc"))
    # Slice into cube to retrieve data for the inset map showing the
    # data region
    region = cube1[-1, :, :]
    # Average over latitude to reduce cube to 1 dimension
    plot_line = region.collapsed("latitude", iris.analysis.MEAN)

    # Open a window for plotting
    fig = plt.figure()
    # Add a single subplot (axes). Could also use "ax_main = plt.subplot()"
    ax_main = fig.add_subplot(1, 1, 1)
    # Produce a quick plot of the 1D cube
    qplt.plot(plot_line)
```

(continues on next page)

(continued from previous page)

```
    # Set x limits to match the data
    ax_main.set_xlim(0, plot_line.coord("longitude").points.max())
    # Adjust the y limits so that the inset map won't clash with main plot
    ax_main.set_ylim(294, 310)
    ax_main.set_title("Meridional Mean Temperature")
    # Add grid lines
    ax_main.grid()

    # Add a second set of axes specifying the fractional coordinates within
    # the figure with bottom left corner at x=0.55, y=0.58 with width
    # 0.3 and height 0.25.
    # Also specify the projection
    ax_sub = fig.add_axes(
        [0.55, 0.58, 0.3, 0.25],
        projection=ccrs.Mollweide(central_longitude=180),
    )

    # Use iris.plot (iplt) here so colour bar properties can be specified
    # Also use a sequential colour scheme to reduce confusion for those with
    # colour-blindness
    iplt.pcolormesh(region, cmap="Blues")
    # Manually set the orientation and tick marks on your colour bar
    ticklist = np.linspace(np.min(region.data), np.max(region.data), 4)
    plt.colorbar(orientation="horizontal", ticks=ticklist)
    ax_sub.set_title("Data Region")
    # Add coastlines
    ax_sub.coastlines()
    # request to show entire map, using the colour mesh on the data region only
    ax_sub.set_global()

    qplt.show()


if __name__ == "__main__":
    main()
```

Total running time of the script: (0 minutes 0.407 seconds)

2.1.8 Applying a Filter to a Time-Series

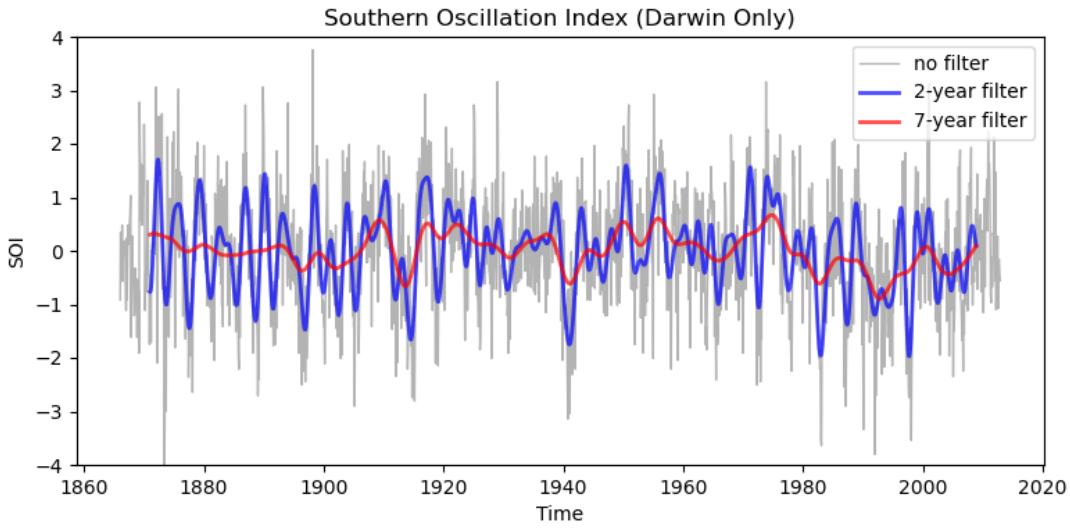
This example demonstrates low pass filtering a time-series by applying a weighted running mean over the time dimension.

The time-series used is the Darwin-only Southern Oscillation index (SOI), which is filtered using two different Lanczos filters, one to filter out time-scales of less than two years and one to filter out time-scales of less than 7 years.

References

Duchon C. E. (1979) Lanczos Filtering in One and Two Dimensions. Journal of Applied Meteorology, Vol 18, pp 1016-1022.

Trenberth K. E. (1984) Signal Versus Noise in the Southern Oscillation. Monthly Weather Review, Vol 112, pp 326-332




```
import matplotlib.pyplot as plt
import numpy as np

import iris
import iris.plot as iplt


def low_pass_weights(window, cutoff):
    """Calculate weights for a low pass Lanczos filter.

    Args:

    window: int
        The length of the filter window.

    cutoff: float
        The cutoff frequency in inverse time steps.

    """
    order = ((window - 1) // 2) + 1
    nwts = 2 * order + 1
    w = np.zeros([nwts])
    n = nwts // 2
    w[n] = 2 * cutoff
    k = np.arange(1.0, n)
    sigma = np.sin(np.pi * k / n) * n / (np.pi * k)
    firstfactor = np.sin(2.0 * np.pi * cutoff * k) / (np.pi * k)
    w[n - 1 : 0 : -1] = firstfactor * sigma
```

(continues on next page)

(continued from previous page)

```
    w[n + 1 : -1] = firstfactor * sigma
    return w[1:-1]


def main():
    # Load the monthly-valued Southern Oscillation Index (SOI) time-series.
    fname = iris.sample_data_path("SOI_Darwin.nc")
    soi = iris.load_cube(fname)

    # Window length for filters.
    window = 121

    # Construct 2-year (24-month) and 7-year (84-month) low pass filters
    # for the SOI data which is monthly.
    wgts24 = low_pass_weights(window, 1.0 / 24.0)
    wgts84 = low_pass_weights(window, 1.0 / 84.0)

    # Apply each filter using the rolling_window method used with the weights
    # keyword argument. A weighted sum is required because the magnitude of
    # the weights are just as important as their relative sizes.
    soi24 = soi.rolling_window(
        "time", iris.analysis.SUM, len(wgts24), weights=wgts24
    )
    soi84 = soi.rolling_window(
        "time", iris.analysis.SUM, len(wgts84), weights=wgts84
    )

    # Plot the SOI time series and both filtered versions.
    plt.figure(figsize=(9, 4))
    iplt.plot(
        soi,
        color="0.7",
        linewidth=1.0,
        linestyle="-",
        alpha=1.0,
        label="no filter",
    )
    iplt.plot(
        soi24,
        color="b",
        linewidth=2.0,
        linestyle="-",
        alpha=0.7,
        label="2-year filter",
    )
    iplt.plot(
        soi84,
        color="r",
        linewidth=2.0,
        linestyle="-",
        alpha=0.7,
        label="7-year filter",
```

(continues on next page)

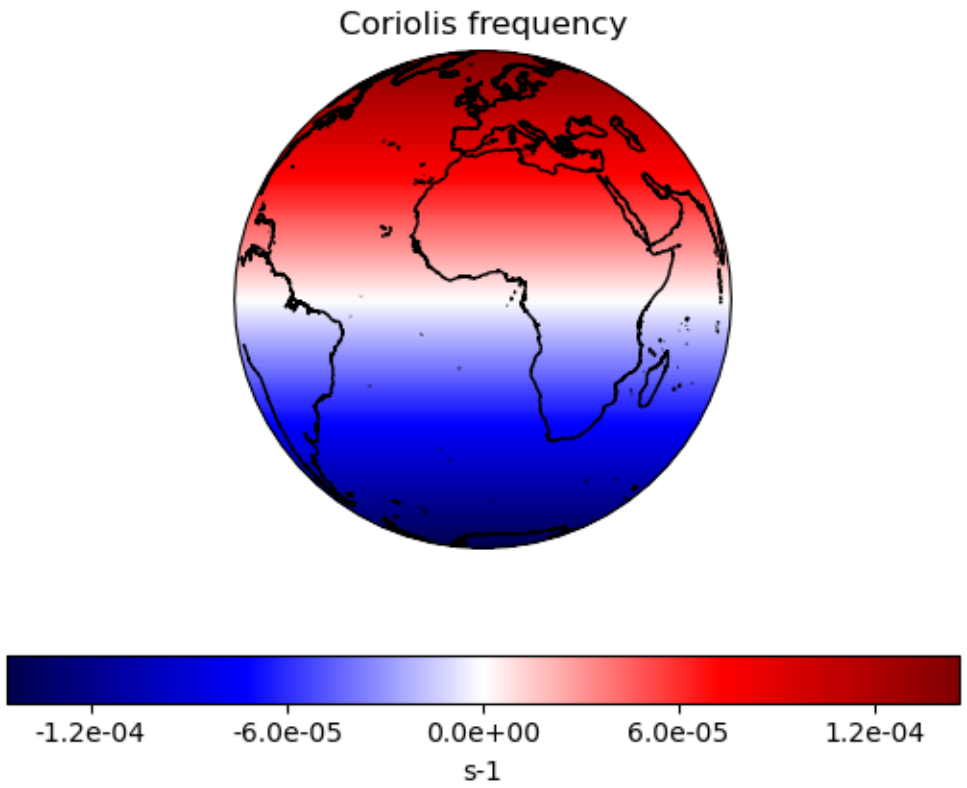
(continued from previous page)

```
    )
    plt.ylim([-4, 4])
    plt.title("Southern Oscillation Index (Darwin Only)")
    plt.xlabel("Time")
    plt.ylabel("SOI")
    plt.legend(fontsize=10)
    iplt.show()


if __name__ == "__main__":
    main()
```

Total running time of the script: (0 minutes 0.770 seconds)

2.1.9 Deriving the Coriolis Frequency Over the Globe

This code computes the Coriolis frequency and stores it in a cube with associated metadata. It then plots the Coriolis frequency on an orthographic projection.




```
import cartopy.crs as ccrs
import matplotlib.pyplot as plt
import numpy as np
```

(continues on next page)

(continued from previous page)

```
import iris
from iris.coord_systems import GeogCS
import iris.plot as iplt


def main():
    # Start with arrays for latitudes and longitudes, with a given number of
    # coordinates in the arrays.
    coordinate_points = 200
    longitudes = np.linspace(-180.0, 180.0, coordinate_points)
    latitudes = np.linspace(-90.0, 90.0, coordinate_points)
    lon2d, lat2d = np.meshgrid(longitudes, latitudes)

    # Omega is the Earth's rotation rate, expressed in radians per second
    omega = 7.29e-5

    # The data for our cube is the Coriolis frequency,
    # `f = 2 * omega * sin(phi)`, which is computed for each grid point over
    # the globe from the 2-dimensional latitude array.
    data = 2.0 * omega * np.sin(np.deg2rad(lat2d))

    # We now need to define a coordinate system for the plot.
    # Here we'll use GeogCS; 6371229 is the radius of the Earth in metres.
    cs = GeogCS(6371229)

    # The Iris coords module turns the latitude list into a coordinate array.
    # Coords then applies an appropriate standard name and unit to it.
    lat_coord = iris.coords.DimCoord(
        latitudes, standard_name="latitude", units="degrees", coord_system=cs
    )

    # The above process is repeated for the longitude coordinates.
    lon_coord = iris.coords.DimCoord(
        longitudes, standard_name="longitude", units="degrees", coord_system=cs
    )

    # Now we add bounds to our latitude and longitude coordinates.
    # We want simple, contiguous bounds for our regularly-spaced coordinate
    # points so we use the guess_bounds() method of the coordinate. For more
    # complex coordinates, we could derive and set the bounds manually.
    lat_coord.guess_bounds()
    lon_coord.guess_bounds()

    # Now we input our data array into the cube.
    new_cube = iris.cube.Cube(
        data,
        standard_name="coriolis_parameter",
        units="s-1",
        dim_coords_and_dims=[(lat_coord, 0), (lon_coord, 1)],
    )
```

(continues on next page)

(continued from previous page)

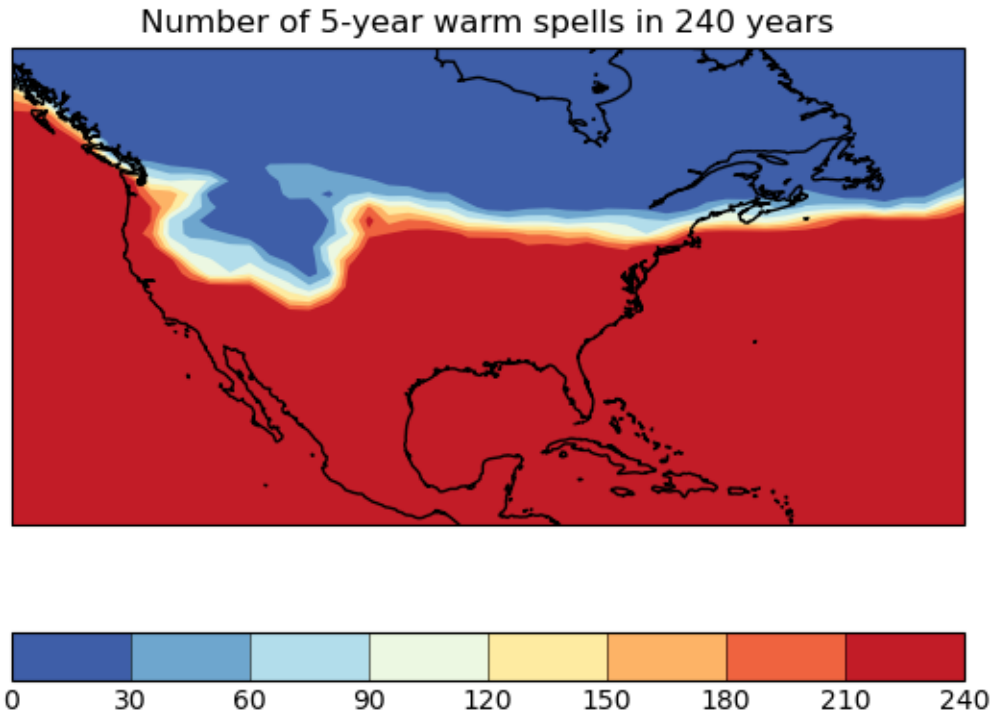
```
    # Now let's plot our cube, along with coastlines, a title and an
    # appropriately-labelled colour bar:
    ax = plt.axes(projection=ccrs.Orthographic())
    ax.coastlines(resolution="10m")
    mesh = iplt.pcolormesh(new_cube, cmap="seismic")
    tick_levels = [-0.00012, -0.00006, 0.0, 0.00006, 0.00012]
    plt.colorbar(
        mesh,
        orientation="horizontal",
        label="s-1",
        ticks=tick_levels,
        format="%.1e",
    )
    plt.title("Coriolis frequency")
    plt.show()


if __name__ == "__main__":
    main()
```

Total running time of the script: (0 minutes 2.169 seconds)

2.1.10 Calculating a Custom Statistic

This example shows how to define and use a custom *iris.analysis.Aggregator*, that provides a new statistical operator for use with cube aggregation functions such as *collapsed()*, *aggregated_by()* or *rolling_window()*.

In this case, we have a 240-year sequence of yearly average surface temperature over North America, and we want to calculate in how many years these exceed a certain temperature over a spell of 5 years or more.




```
import matplotlib.pyplot as plt
import numpy as np

import iris
from iris.analysis import Aggregator
import iris.plot as iplt
import iris.quickplot as qplt
from iris.util import rolling_window


# Define a function to perform the custom statistical operation.
# Note: in order to meet the requirements of iris.analysis.Aggregator, it must
# do the calculation over an arbitrary (given) data axis.
def count_spells(data, threshold, axis, spell_length):
    """
    Function to calculate the number of points in a sequence where the value
    has exceeded a threshold value for at least a certain number of timepoints.

    Generalised to operate on multiple time sequences arranged on a specific
    axis of a multidimensional array.

    Args:
```

(continues on next page)

(continued from previous page)

```
    * data (array):
        raw data to be compared with value threshold.

    * threshold (float):
        threshold point for 'significant' datapoints.

    * axis (int):
        number of the array dimension mapping the time sequences.
        (Can also be negative, e.g. '-1' means last dimension)

    * spell_length (int):
        number of consecutive times at which value > threshold to "count".

    """
    if axis < 0:
        # just cope with negative axis numbers
        axis += data.ndim
    # Threshold the data to find the 'significant' points.
    data_hits = data > threshold
    # Make an array with data values "windowed" along the time axis.
    hit_windows = rolling_window(data_hits, window=spell_length, axis=axis)
    # Find the windows "full of True-s" (along the added 'window axis').
    full_windows = np.all(hit_windows, axis=axis + 1)
    # Count points fulfilling the condition (along the time axis).
    spell_point_counts = np.sum(full_windows, axis=axis, dtype=int)
    return spell_point_counts


def main():
    # Load the whole time-sequence as a single cube.
    file_path = iris.sample_data_path("E1_north_america.nc")
    cube = iris.load_cube(file_path)

    # Make an aggregator from the user function.
    SPELL_COUNT = Aggregator(
        "spell_count", count_spells, units_func=lambda units: 1
    )

    # Define the parameters of the test.
    threshold_temperature = 280.0
    spell_years = 5

    # Calculate the statistic.
    warm_periods = cube.collapsed(
        "time",
        SPELL_COUNT,
        threshold=threshold_temperature,
        spell_length=spell_years,
    )
    warm_periods.rename("Number of 5-year warm spells in 240 years")

    # Plot the results.
```

(continues on next page)

(continued from previous page)

```
    qplt.contourf(warm_periods, cmap="RdYlBu_r")
    plt.gca().coastlines()
    iplt.show()


if __name__ == "__main__":
    main()
```

Total running time of the script: (0 minutes 0.820 seconds)

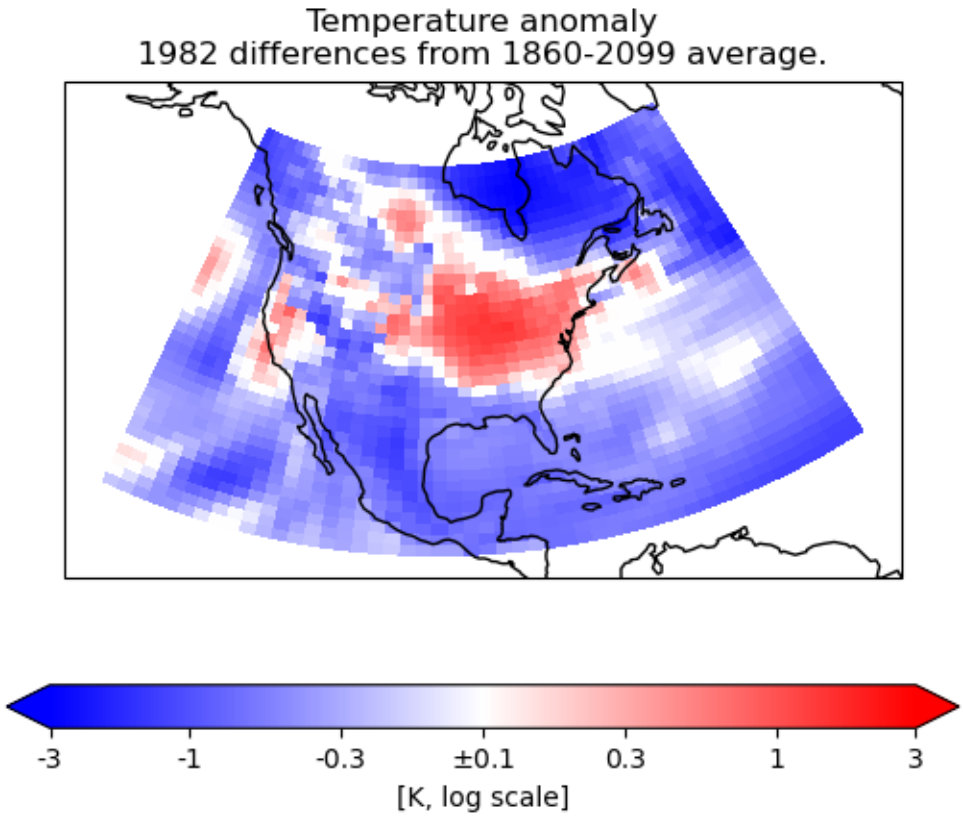
2.1.11 Colouring Anomaly Data With Logarithmic Scaling

In this example, we need to plot anomaly data where the values have a "logarithmic" significance – i.e. we want to give approximately equal ranges of colour between data values of, say, 1 and 10 as between 10 and 100.

As the data range also contains zero, that obviously does not suit a simple logarithmic interpretation. However, values of less than a certain absolute magnitude may be considered "not significant", so we put these into a separate "zero band" which is plotted in white.

To do this, we create a custom value mapping function (normalization) using the matplotlib Norm class `matplotlib.colors.SymLogNorm`. We use this to make a cell-filled pseudocolor plot with a colorbar.

NOTE: By "pseudocolour", we mean that each data point is drawn as a "cell" region on the plot, coloured according to its data value. This is provided in Iris by the functions *`iris.plot.pcolor()`* and *`iris.plot.pcolormesh()`*, which call the underlying matplotlib functions of the same names (i.e., `matplotlib.pyplot.pcolor` and `matplotlib.pyplot.pcolormesh`). See also: http://en.wikipedia.org/wiki/False_color#Pseudocolor.




```
import cartopy.crs as ccrs
import matplotlib.colors as mcols
import matplotlib.pyplot as plt

import iris
import iris.coord_categorisation
import iris.plot as iplt


def main():
    # Load a sample air temperatures sequence.
    file_path = iris.sample_data_path("E1_north_america.nc")
    temperatures = iris.load_cube(file_path)

    # Create a year-number coordinate from the time information.
    iris.coord_categorisation.add_year(temperatures, "time")

    # Create a sample anomaly field for one chosen year, by extracting that
    # year and subtracting the time mean.
    sample_year = 1982
    year_temperature = temperatures.extract(iris.Constraint(year=sample_year))
    time_mean = temperatures.collapsed("time", iris.analysis.MEAN)
    anomaly = year_temperature - time_mean
```

(continues on next page)

(continued from previous page)

```
    # Construct a plot title string explaining which years are involved.
    years = temperatures.coord("year").points
    plot_title = "Temperature anomaly"
    plot_title += "\n{} differences from {}-{} average.".format(
        sample_year, years[0], years[-1]
    )

    # Define scaling levels for the logarithmic colouring.
    minimum_log_level = 0.1
    maximum_scale_level = 3.0

    # Use a standard colour map which varies blue-white-red.
    # For suitable options, see the 'Diverging colormaps' section in:
    # http://matplotlib.org/stable/gallery/color/colormap_reference.html
    anom_cmap = "bwr"

    # Create a 'logarithmic' data normalization.
    anom_norm = mcols.SymLogNorm(
        linthresh=minimum_log_level,
        linscale=0.01,
        vmin=-maximum_scale_level,
        vmax=maximum_scale_level,
    )
    # Setting "linthresh=minimum_log_level" makes its non-logarithmic
    # data range equal to our 'zero band'.
    # Setting "linscale=0.01" maps the whole zero band to the middle colour value
    # (i.e., 0.5), which is the neutral point of a "diverging" style colormap.

    # Create an Axes, specifying the map projection.
    plt.axes(projection=ccrs.LambertConformal())

    # Make a pseudocolour plot using this colour scheme.
    mesh = iplt.pcolormesh(anomaly, cmap=anom_cmap, norm=anom_norm)

    # Add a colourbar, with extensions to show handling of out-of-range values.
    bar = plt.colorbar(mesh, orientation="horizontal", extend="both")

    # Set some suitable fixed "logarithmic" colourbar tick positions.
    tick_levels = [-3, -1, -0.3, 0.0, 0.3, 1, 3]
    bar.set_ticks(tick_levels)

    # Modify the tick labels so that the centre one shows "+/-<minumum-level>".
    tick_levels[3] = r"$\pm${:g}".format(minimum_log_level)
    bar.set_ticklabels(tick_levels)

    # Label the colourbar to show the units.
    bar.set_label("[{}, log scale]".format(anomaly.units))

    # Add coastlines and a title.
    plt.gca().coastlines()
    plt.title(plot_title)
```

(continues on next page)

(continued from previous page)

```
    # Display the result.
    iplt.show()


if __name__ == "__main__":
    main()
```

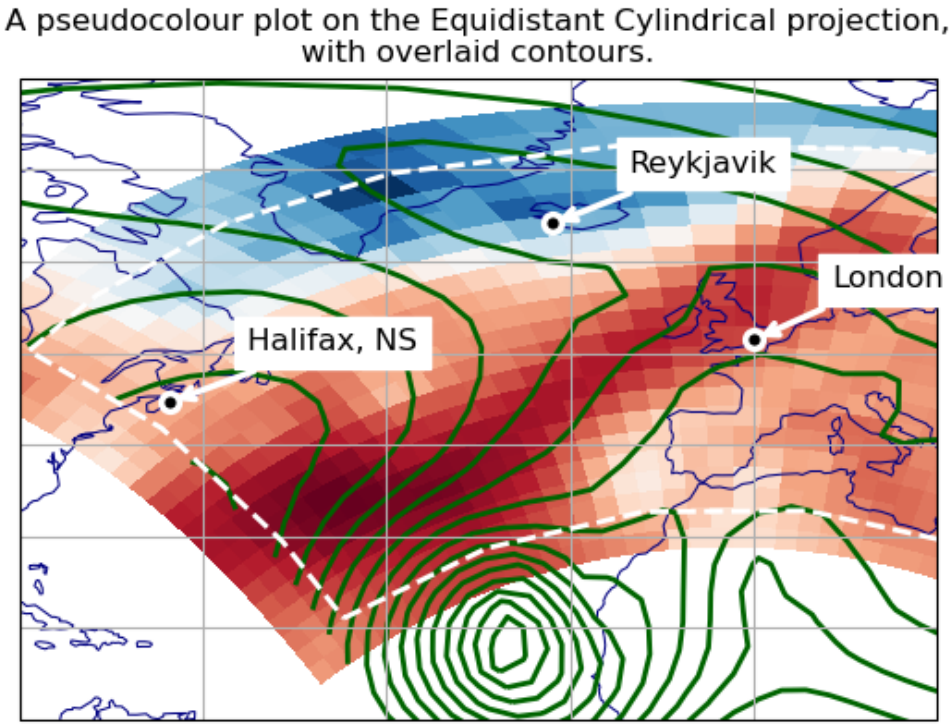
Total running time of the script: (0 minutes 0.387 seconds)

2.1.12 Plotting in Different Projections

This example shows how to overlay data and graphics in different projections, demonstrating various features of Iris, Cartopy and matplotlib.

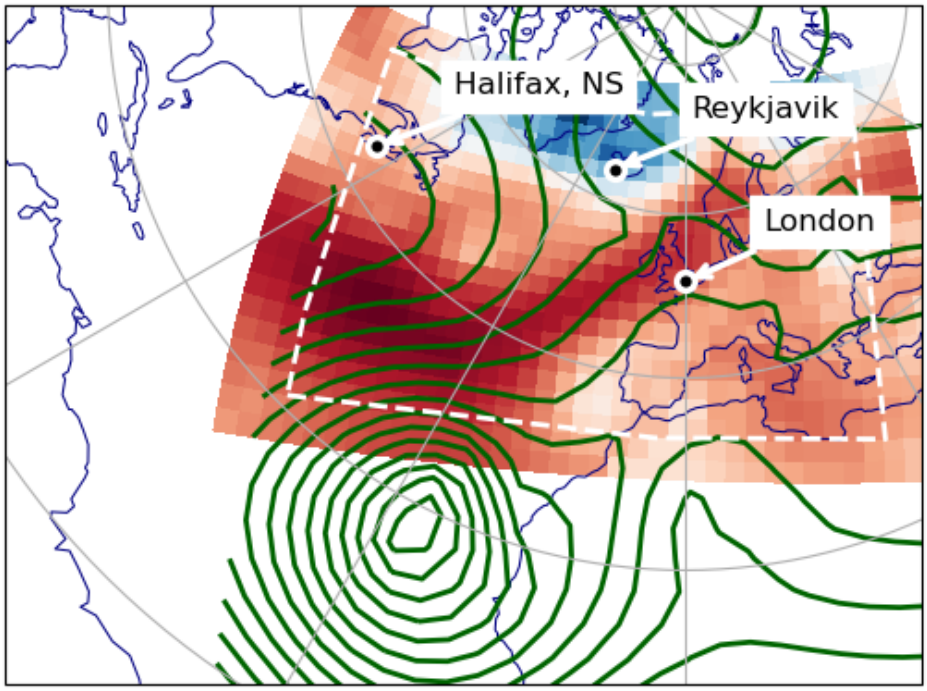
We wish to overlay two datasets, defined on different rotated-pole grids. To display both together, we make a pseudo-coloured plot of the first, overlaid with contour lines from the second. We also add some lines and text annotations drawn in various projections.

We plot these over a specified region, in two different map projections.



-

A pseudocolour plot on the North Polar Stereographic projection, with overlaid contours.



-

```
import cartopy.crs as ccrs
import matplotlib.pyplot as plt
import numpy as np

import iris
import iris.plot as iplt

# Define a Cartopy 'ordinary' lat-lon coordinate reference system.
crs_latlon = ccrs.PlateCarree()


def make_plot(projection_name, projection_crs):

    # Create a matplotlib Figure.
    plt.figure()

    # Add a matplotlib Axes, specifying the required display projection.
    # NOTE: specifying 'projection' (a "cartopy.crs.Projection") makes the
    # resulting Axes a "cartopy.mpl.geoaxes.GeoAxes", which supports plotting
    # in different coordinate systems.
    ax = plt.axes(projection=projection_crs)

    # Set display limits to include a set region of latitude * longitude.
    # (Note: Cartopy-specific).
```

(continues on next page)

(continued from previous page)

```
    ax.set_extent((-80.0, 20.0, 10.0, 80.0), crs=crs_latlon)

    # Add coastlines and meridians/parallels (Cartopy-specific).
    ax.coastlines(linewidth=0.75, color="navy")
    ax.gridlines(crs=crs_latlon, linestyle="-")

    # Plot the first dataset as a pseudocolour filled plot.
    maindata_filepath = iris.sample_data_path("rotated_pole.nc")
    main_data = iris.load_cube(maindata_filepath)
    # NOTE: iplt.pcolormesh calls "pyplot.pcolormesh", passing in a coordinate
    # system with the 'transform' keyword:  This enables the Axes (a cartopy
    # GeoAxes) to reproject the plot into the display projection.
    iplt.pcolormesh(main_data, cmap="RdBu_r")

    # Overplot the other dataset (which has a different grid), as contours.
    overlay_filepath = iris.sample_data_path("space_weather.nc")
    overlay_data = iris.load_cube(overlay_filepath, "total electron content")
    # NOTE: as above, "iris.plot.contour" calls "pyplot.contour" with a
    # 'transform' keyword, enabling Cartopy reprojection.
    iplt.contour(
        overlay_data, 20, linewidths=2.0, colors="darkgreen", linestyles="-"
    )

    # Draw a margin line, some way in from the border of the 'main' data...
    # First calculate rectangle corners, 7% in from each corner of the data.
    x_coord, y_coord = main_data.coord(axis="x"), main_data.coord(axis="y")
    x_start, x_end = np.min(x_coord.points), np.max(x_coord.points)
    y_start, y_end = np.min(y_coord.points), np.max(y_coord.points)
    margin = 0.07
    margin_fractions = np.array([margin, 1.0 - margin])
    x_lower, x_upper = x_start + (x_end - x_start) * margin_fractions
    y_lower, y_upper = y_start + (y_end - y_start) * margin_fractions
    box_x_points = x_lower + (x_upper - x_lower) * np.array([0, 1, 1, 0, 0])
    box_y_points = y_lower + (y_upper - y_lower) * np.array([0, 0, 1, 1, 0])
    # Get the Iris coordinate sytem of the X coordinate (Y should be the same).
    cs_data1 = x_coord.coord_system
    # Construct an equivalent Cartopy coordinate reference system ("crs").
    crs_data1 = cs_data1.as_cartopy_crs()
    # Draw the rectangle in this crs, with matplotlib "pyplot.plot".
    # NOTE: the 'transform' keyword specifies a non-display coordinate system
    # for the plot points (as used by the "iris.plot" functions).
    plt.plot(
        box_x_points,
        box_y_points,
        transform=crs_data1,
        linewidth=2.0,
        color="white",
        linestyle="--",
    )

    # Mark some particular places with a small circle and a name label...
    # Define some test points with latitude and longitude coordinates.
```

(continues on next page)

(continued from previous page)

```
    city_data = [
        ("London", 51.5072, 0.1275),
        ("Halifax, NS", 44.67, -63.61),
        ("Reykjavik", 64.1333, -21.9333),
    ]
    # Place a single marker point and a text annotation at each place.
    for name, lat, lon in city_data:
        plt.plot(
            lon,
            lat,
            marker="o",
            markersize=7.0,
            markeredgewidth=2.5,
            markerfacecolor="black",
            markeredgecolor="white",
            transform=crs_latlon,
        )
        # NOTE: the "plt.annotate call" does not have a "transform=" keyword,
        # so for this one we transform the coordinates with a Cartopy call.
        at_x, at_y = ax.projection.transform_point(
            lon, lat, src_crs=crs_latlon
        )
        plt.annotate(
            name,
            xy=(at_x, at_y),
            xytext=(30, 20),
            textcoords="offset points",
            color="black",
            backgroundcolor="white",
            size="large",
            arrowprops=dict(arrowstyle="->", color="white", linewidth=2.5),
        )

    # Add a title, and display.
    plt.title(
        "A pseudocolour plot on the {} projection,\n"
        "with overlaid contours.".format(projection_name)
    )
    iplt.show()


def main():
    # Demonstrate with two different display projections.
    make_plot("Equidistant Cylindrical", ccrs.PlateCarree())
    make_plot("North Polar Stereographic", ccrs.NorthPolarStereo())


if __name__ == "__main__":
    main()
```

Total running time of the script: (0 minutes 0.470 seconds)

2.1.13 Loading a Cube From a Custom File Format

This example shows how a custom text file can be loaded using the standard Iris load mechanism.

The first stage in the process is to define an Iris *FormatSpecification* for the file format. To create a format specification we need to define the following:

- format_name - Some text that describes the format specification we are creating
- file_element - FileElement object describing the element which identifies this FormatSpecification.

  Possible values are:

  iris.io.format_picker.MagicNumber(n, o) The n bytes from the file at offset o.

  iris.io.format_picker.FileExtension() The file's extension.

  iris.io.format_picker.LeadingLine() The first line of the file.
- file_element_value - The value that the file_element should take if a file matches this FormatSpecification
- handler (optional) - A generator function that will be called when the file specification has been identified. This function is provided by the user and provides the means to parse the whole file. If no handler function is provided, then identification is still possible without any handling.

  The handler function must define the following arguments:

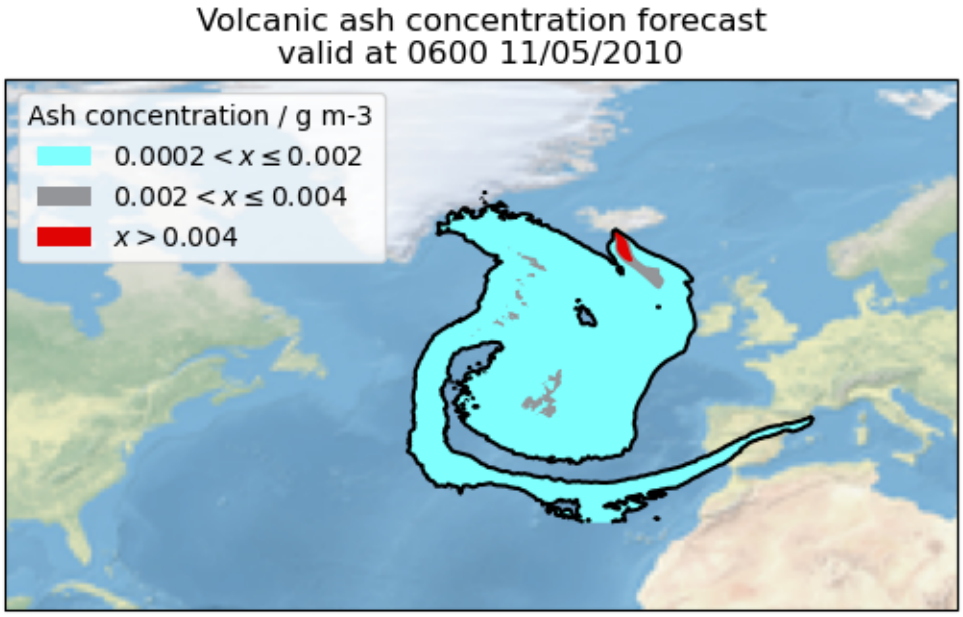
  – list of filenames to process

  – callback function - An optional function to filter/alter the Iris cubes returned

  The handler function must be defined as generator which yields each cube as they are produced.
- priority (optional) - Integer giving a priority for considering this specification where higher priority means sooner consideration

In the following example, the function `load_NAME_III()` has been defined to handle the loading of the raw data from the custom file format. This function is called from `NAME_to_cube()` which uses this data to create and yield Iris cubes.

In the `main()` function the filenames are loaded via the `iris.load_cube` function which automatically invokes the `FormatSpecification` we defined. The cube returned from the load function is then used to produce a plot.




```
import datetime

from cf_units import CALENDAR_GREGORIAN, Unit
import matplotlib.pyplot as plt
import numpy as np

import iris
import iris.coord_systems as icoord_systems
import iris.coords as icoords
import iris.fileformats
import iris.io.format_picker as format_picker
import iris.plot as iplt

UTC_format = "%H%M%Z %d/%m/%Y"

FLOAT_HEADERS = [
    "X grid origin",
    "Y grid origin",
    "X grid resolution",
    "Y grid resolution",
]
INT_HEADERS = ["X grid size", "Y grid size", "Number of fields"]
DATE_HEADERS = ["Run time", "Start of release", "End of release"]
```

(continues on next page)

(continued from previous page)

```
COLUMN_NAMES = [
    "species_category",
    "species",
    "cell_measure",
    "quantity",
    "unit",
    "z_level",
    "time",
]


def load_NAME_III(filename):
    """
    Loads the Met Office's NAME III grid output files returning headers, column
    definitions and data arrays as 3 separate lists.

    """

    # Loading a file gives a generator of lines which can be progressed using
    # the next() function. This will come in handy as we wish to progress
    # through the file line by line.
    with open(filename) as file_handle:
        # Define a dictionary which can hold the header metadata for this file.
        headers = {}

        # Skip the NAME header of the file which looks something like
        # 'NAME III (version X.X.X)'.
        next(file_handle)

        # Read the next 16 lines of header information, putting the form
        # "header name:    header value" into a dictionary.
        for _ in range(16):
            header_name, header_value = next(file_handle).split(":")

            # Strip off any spurious space characters in the header name and
            # value.
            header_name = header_name.strip()
            header_value = header_value.strip()

            # Cast some headers into floats or integers if they match a given
            # header name.
            if header_name in FLOAT_HEADERS:
                header_value = float(header_value)
            elif header_name in INT_HEADERS:
                header_value = int(header_value)
            elif header_name in DATE_HEADERS:
                # convert the time to python datetimes
                header_value = datetime.datetime.strptime(
                    header_value, UTC_format
                )

            headers[header_name] = header_value
```

(continues on next page)

(continued from previous page)

```
        # Skip the next blank line in the file.
        next(file_handle)

        # Read the next 7 lines of column definitions.
        column_headings = {}
        for column_header_name in COLUMN_NAMES:
            column_headings[column_header_name] = [
                col.strip() for col in next(file_handle).split(",")
            ][:-1]

        # Convert the time to python datetimes.
        new_time_column_header = []
        for i, t in enumerate(column_headings["time"]):
            # The first 4 columns aren't time at all, so don't convert them to
            # datetimes.
            if i >= 4:
                t = datetime.datetime.strptime(t, UTC_format)
            new_time_column_header.append(t)
        column_headings["time"] = new_time_column_header

        # Skip the blank line after the column headers.
        next(file_handle)

        # Make a list of data arrays to hold the data for each column.
        data_shape = (headers["Y grid size"], headers["X grid size"])
        data_arrays = [
            np.zeros(data_shape, dtype=np.float32)
            for i in range(headers["Number of fields"])
        ]

        # Iterate over the remaining lines which represent the data in a column
        # form.
        for line in file_handle:
            # Split the line by comma, removing the last empty column caused by
            # the trailing comma.
            vals = line.split(",")[:-1]

            # Cast the x and y grid positions to floats and convert them to
            # zero based indices (the numbers are 1 based grid positions where
            # 0.5 represents half a grid point.)
            x = int(float(vals[0]) - 1.5)
            y = int(float(vals[1]) - 1.5)

            # Populate the data arrays (i.e. all columns but the leading 4).
            for i, data_array in enumerate(data_arrays):
                data_array[y, x] = float(vals[i + 4])

    return headers, column_headings, data_arrays


def NAME_to_cube(filenames, callback):
```

(continues on next page)

(continued from previous page)

```
    """
    Returns a generator of cubes given a list of filenames and a callback.
    """

    for filename in filenames:
        header, column_headings, data_arrays = load_NAME_III(filename)

        for i, data_array in enumerate(data_arrays):
            # turn the dictionary of column headers with a list of header
            # information for each field into a dictionary of headers for just
            # this field. Ignore the first 4 columns of grid position (data was
            # located with the data array).
            field_headings = dict(
                (k, v[i + 4]) for k, v in column_headings.items()
            )

            # make an cube
            cube = iris.cube.Cube(data_array)

            # define the name and unit
            name = "%s %s" % (
                field_headings["species"],
                field_headings["quantity"],
            )
            name = name.upper().replace(" ", "_")
            cube.rename(name)
            # Some units are badly encoded in the file, fix this by putting a
            # space in between. (if gs is not found, then the string will be
            # returned unchanged)
            cube.units = field_headings["unit"].replace("gs", "g s")

            # define and add the singular coordinates of the field (flight
            # level, time etc.)
            cube.add_aux_coord(
                icoords.AuxCoord(
                    field_headings["z_level"],
                    long_name="flight_level",
                    units="1",
                )
            )

            # define the time unit and use it to serialise the datetime for the
            # time coordinate
            time_unit = Unit("hours since epoch", calendar=CALENDAR_GREGORIAN)
            time_coord = icoords.AuxCoord(
                time_unit.date2num(field_headings["time"]),
                standard_name="time",
                units=time_unit,
            )
            cube.add_aux_coord(time_coord)

            # build a coordinate system which can be referenced by latitude and
```

(continues on next page)

(continued from previous page)

```
            # longitude coordinates
            lat_lon_coord_system = icoord_systems.GeogCS(6371229)

            # build regular latitude and longitude coordinates which have
            # bounds
            start = header["X grid origin"] + header["X grid resolution"]
            step = header["X grid resolution"]
            count = header["X grid size"]
            pts = start + np.arange(count, dtype=np.float32) * step
            lon_coord = icoords.DimCoord(
                pts,
                standard_name="longitude",
                units="degrees",
                coord_system=lat_lon_coord_system,
            )
            lon_coord.guess_bounds()

            start = header["Y grid origin"] + header["Y grid resolution"]
            step = header["Y grid resolution"]
            count = header["Y grid size"]
            pts = start + np.arange(count, dtype=np.float32) * step
            lat_coord = icoords.DimCoord(
                pts,
                standard_name="latitude",
                units="degrees",
                coord_system=lat_lon_coord_system,
            )
            lat_coord.guess_bounds()

            # add the latitude and longitude coordinates to the cube, with
            # mappings to data dimensions
            cube.add_dim_coord(lat_coord, 0)
            cube.add_dim_coord(lon_coord, 1)

            # implement standard iris callback capability. Although callbacks
            # are not used in this example, the standard mechanism for a custom
            # loader to implement a callback is shown:
            cube = iris.io.run_callback(
                callback, cube, [header, field_headings, data_array], filename
            )

            # yield the cube created (the loop will continue when the next()
            # element is requested)
            yield cube


# Create a format_picker specification of the NAME file format giving it a
# priority greater than the built in NAME loader.
_NAME_III_spec = format_picker.FormatSpecification(
    "Name III",
    format_picker.LeadingLine(),
    lambda line: line.startswith(b"NAME III"),
```

(continues on next page)

(continued from previous page)

```
    NAME_to_cube,
    priority=6,
)

# Register the NAME loader with iris
iris.fileformats.FORMAT_AGENT.add_spec(_NAME_III_spec)


# --------------------------------------------
# |          Using the new loader            |
# --------------------------------------------


def main():
    fname = iris.sample_data_path("NAME_output.txt")

    boundary_volc_ash_constraint = iris.Constraint(
        "VOLCANIC_ASH_AIR_CONCENTRATION", flight_level="From FL000 - FL200"
    )

    # Callback shown as None to illustrate where a cube-level callback function
    # would be used if required
    cube = iris.load_cube(fname, boundary_volc_ash_constraint, callback=None)

    # draw contour levels for the data (the top level is just a catch-all)
    levels = (0.0002, 0.002, 0.004, 1e10)
    cs = iplt.contourf(
        cube,
        levels=levels,
        colors=("#80ffff", "#939598", "#e00404"),
    )

    # draw a black outline at the lowest contour to highlight affected areas
    iplt.contour(cube, levels=(levels[0], 100), colors="black")

    # set an extent and a background image for the map
    ax = plt.gca()
    ax.set_extent((-90, 20, 20, 75))
    ax.stock_img("ne_shaded")

    # make a legend, with custom labels, for the coloured contour set
    artists, _ = cs.legend_elements()
    labels = [
        r"$%s < x \leq %s$" % (levels[0], levels[1]),
        r"$%s < x \leq %s$" % (levels[1], levels[2]),
        r"$x > %s$" % levels[2],
    ]
    ax.legend(
        artists, labels, title="Ash concentration / g m-3", loc="upper left"
    )

    time = cube.coord("time")
```

(continues on next page)

(continued from previous page)

```
    time_date = time.units.num2date(time.points[0]).strftime(UTC_format)
    plt.title("Volcanic ash concentration forecast\nvalid at %s" % time_date)

    iplt.show()


if __name__ == "__main__":
    main()
```

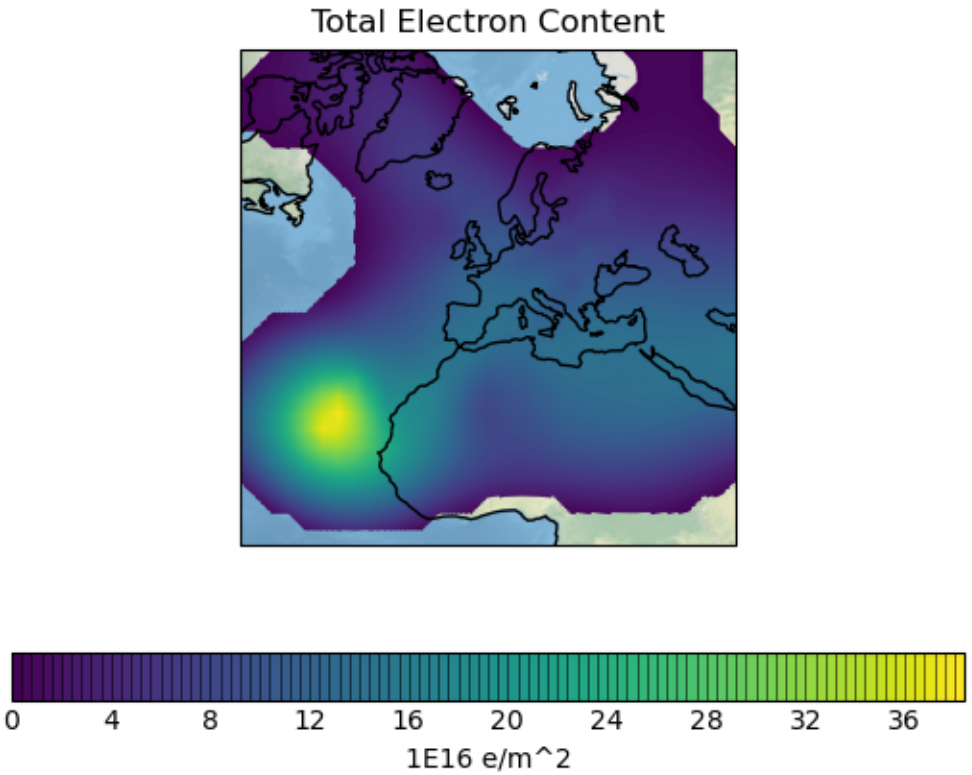
Total running time of the script: (0 minutes 0.380 seconds)

2.2 Meteorology

2.2.1 Ionosphere Space Weather

This space weather example plots a filled contour of rotated pole point data with a shaded relief image underlay. The plot shows aggregated vertical electron content in the ionosphere.

The plot exhibits an interesting outline effect due to excluding data values below a certain threshold.

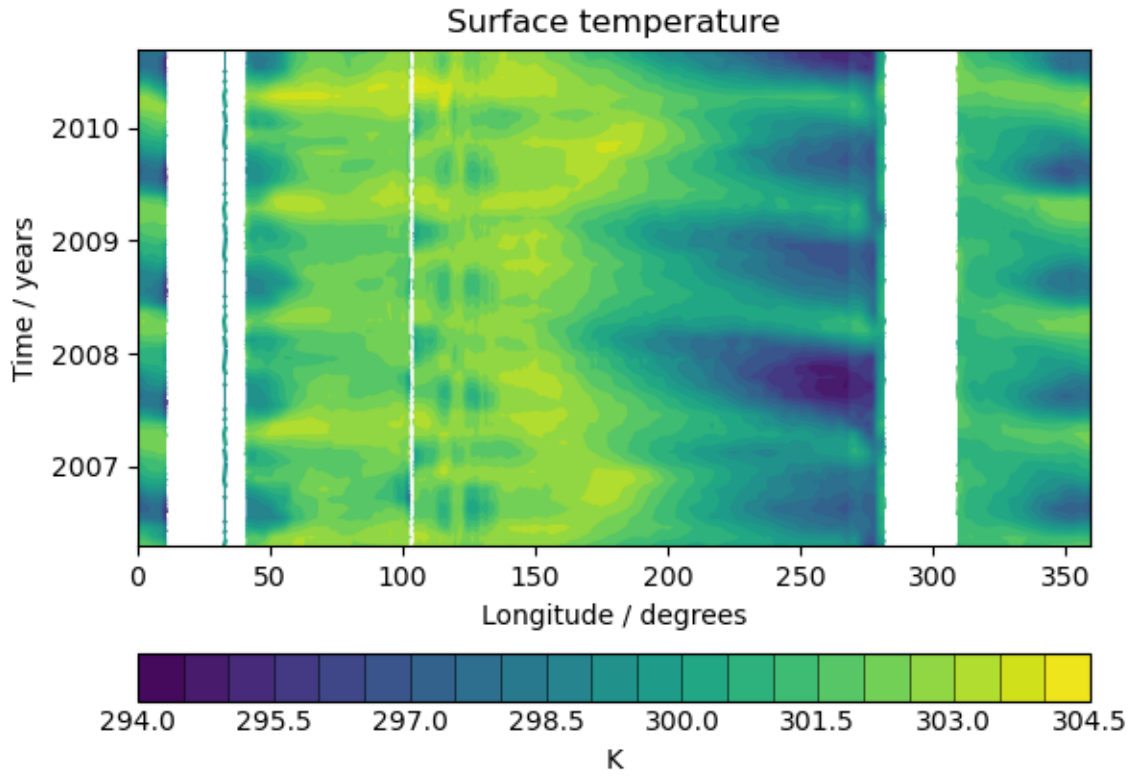


```
import matplotlib.pyplot as plt
import numpy.ma as ma

import iris
import iris.plot as iplt
import iris.quickplot as qplt


def main():
    # Load the "total electron content" cube.
    filename = iris.sample_data_path("space_weather.nc")
    cube = iris.load_cube(filename, "total electron content")

    # Explicitly mask negative electron content.
    cube.data = ma.masked_less(cube.data, 0)

    # Plot the cube using one hundred colour levels.
    qplt.contourf(cube, 100)
    plt.title("Total Electron Content")
    plt.xlabel("longitude / degrees")
    plt.ylabel("latitude / degrees")
    plt.gca().stock_img()
    plt.gca().coastlines()

    iplt.show()


if __name__ == "__main__":
    main()
```

Total running time of the script: (0 minutes 1.468 seconds)

2.2.2 Hovmoller Diagram of Monthly Surface Temperature

This example demonstrates the creation of a Hovmoller diagram with fine control over plot ticks and labels. The data comes from the Met Office OSTIA project and has been pre-processed to calculate the monthly mean sea surface temperature.




```
import matplotlib.dates as mdates
import matplotlib.pyplot as plt

import iris
import iris.plot as iplt
import iris.quickplot as qplt


def main():
    # load a single cube of surface temperature between +/- 5 latitude
    fname = iris.sample_data_path("ostia_monthly.nc")
    cube = iris.load_cube(
        fname,
        iris.Constraint("surface_temperature", latitude=lambda v: -5 < v < 5),
    )

    # Take the mean over latitude
    cube = cube.collapsed("latitude", iris.analysis.MEAN)

    # Now that we have our data in a nice way, lets create the plot
    # contour with 20 levels
    qplt.contourf(cube, 20)
```

(continues on next page)

(continued from previous page)

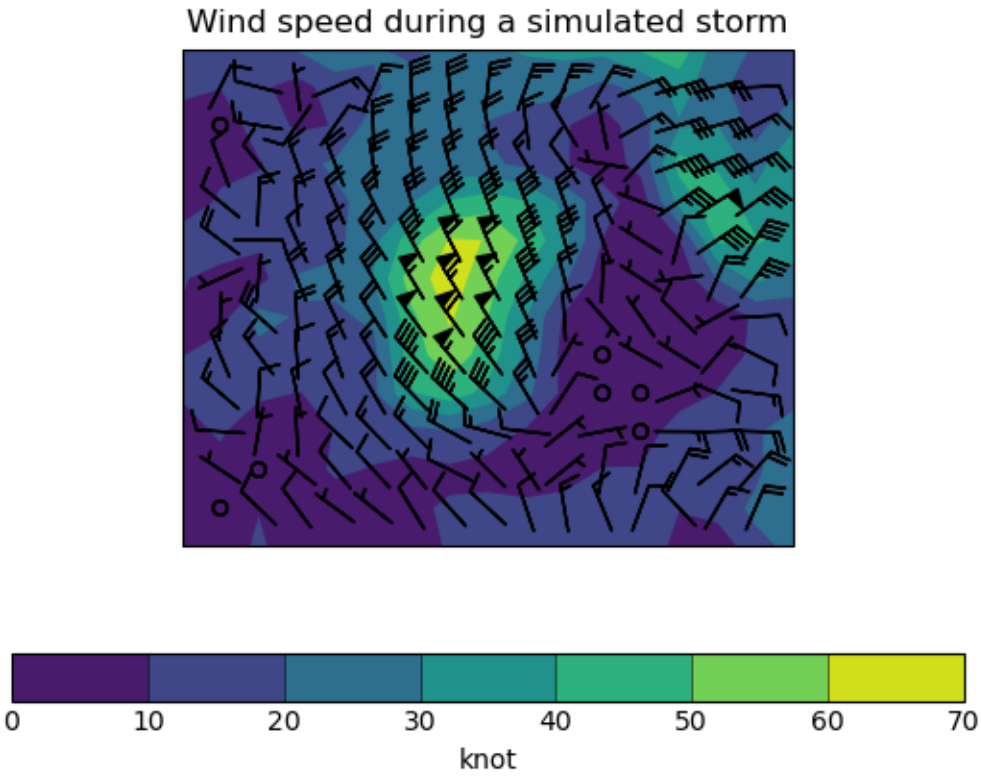
```
    # Put a custom label on the y axis
    plt.ylabel("Time / years")

    # Stop matplotlib providing clever axes range padding
    plt.axis("tight")

    # As we are plotting annual variability, put years as the y ticks
    plt.gca().yaxis.set_major_locator(mdates.YearLocator())

    # And format the ticks to just show the year
    plt.gca().yaxis.set_major_formatter(mdates.DateFormatter("%Y"))

    iplt.show()


if __name__ == "__main__":
    main()
```

Total running time of the script: (0 minutes 0.287 seconds)

2.2.3 Plotting Wind Direction Using Barbs

This example demonstrates using barbs to plot wind speed contours and wind direction barbs from wind vector component input data. The vector components are co-located in space in this case.

The magnitude of the wind in the original data is low and so doesn't illustrate the full range of barbs. The wind is scaled to simulate a storm that better illustrates the range of barbs that are available.




```
import matplotlib.pyplot as plt

import iris
import iris.plot as iplt
import iris.quickplot as qplt


def main():
    # Load the u and v components of wind from a pp file
    infile = iris.sample_data_path("wind_speed_lake_victoria.pp")

    uwind = iris.load_cube(infile, "x_wind")
    vwind = iris.load_cube(infile, "y_wind")

    uwind.convert_units("knot")
    vwind.convert_units("knot")

    # To illustrate the full range of barbs, scale the wind speed up to pretend
    # that a storm is passing over
    magnitude = (uwind ** 2 + vwind ** 2) ** 0.5
    magnitude.convert_units("knot")
    max_speed = magnitude.collapsed(
        ("latitude", "longitude"), iris.analysis.MAX
```

(continues on next page)

(continued from previous page)

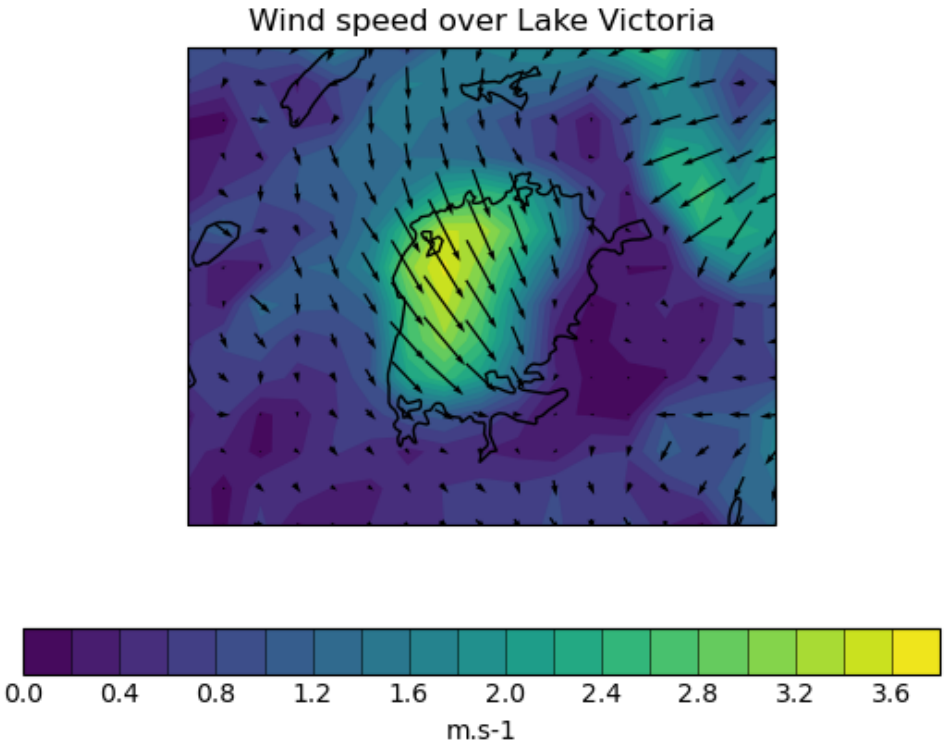
```
    ).data
    max_desired = 65

    uwind = uwind / max_speed * max_desired
    vwind = vwind / max_speed * max_desired

    # Create a cube containing the wind speed
    windspeed = (uwind ** 2 + vwind ** 2) ** 0.5
    windspeed.rename("windspeed")
    windspeed.convert_units("knot")

    plt.figure()

    # Plot the wind speed as a contour plot
    qplt.contourf(windspeed)

    # Add wind barbs except for the outermost values which overhang the edge
    # of the plot if left
    iplt.barbs(uwind[1:-1, 1:-1], vwind[1:-1, 1:-1], pivot="middle", length=6)

    plt.title("Wind speed during a simulated storm")
    qplt.show()


if __name__ == "__main__":
    main()
```

Total running time of the script: (0 minutes 0.192 seconds)

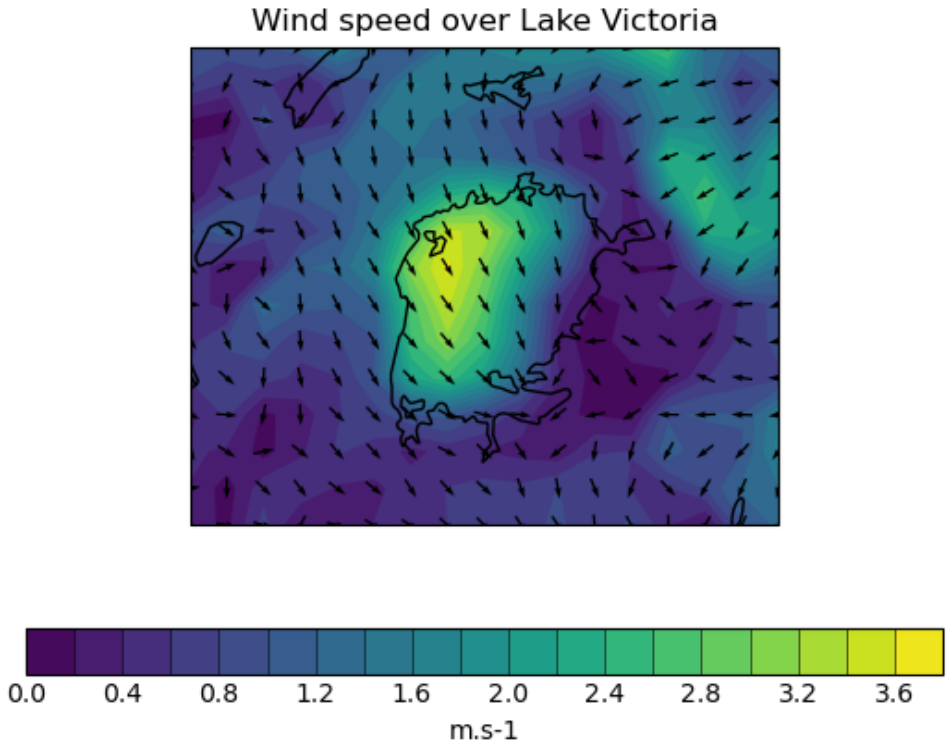
2.2.4 Plotting Wind Direction Using Quiver

This example demonstrates using quiver to plot wind speed contours and wind direction arrows from wind vector component input data. The vector components are co-located in space in this case.

For the second plot, the data used for the arrows is normalised to produce arrows with a uniform size on the plot.


Wind speed over Lake Victoria
0.0
0.4
0.8
1.2
1.6
2.0
2.4
2.8
3.2
3.6
m.s-1

-



-

```
import cartopy.crs as ccrs
import cartopy.feature as cfeat
import matplotlib.pyplot as plt
import numpy as np

import iris
import iris.coord_categorisation
import iris.quickplot as qplt


def main():
    # Load the u and v components of wind from a pp file
    infile = iris.sample_data_path("wind_speed_lake_victoria.pp")

    uwind = iris.load_cube(infile, "x_wind")
    vwind = iris.load_cube(infile, "y_wind")

    ulon = uwind.coord("longitude")
    vlon = vwind.coord("longitude")

    # The longitude points go from 180 to 540, so subtract 360 from them
    ulon.points = ulon.points - 360.0
    vlon.points = vlon.points - 360.0
```

(continues on next page)

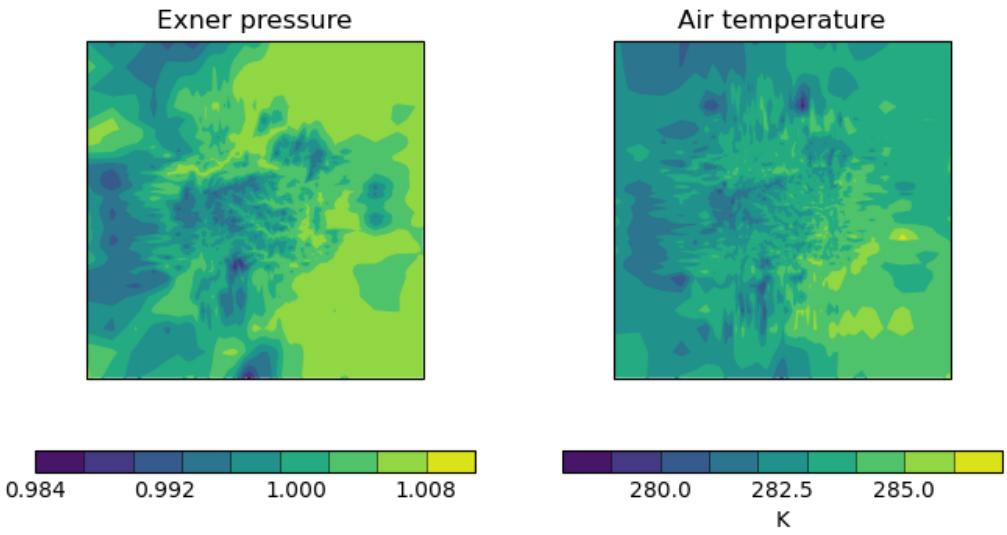
(continued from previous page)

```
    # Create a cube containing the wind speed
    windspeed = (uwind ** 2 + vwind ** 2) ** 0.5
    windspeed.rename("windspeed")

    x = ulon.points
    y = uwind.coord("latitude").points
    u = uwind.data
    v = vwind.data

    # Set up axes to show the lake
    lakes = cfeat.NaturalEarthFeature(
        "physical", "lakes", "50m", facecolor="none"
    )

    plt.figure()
    ax = plt.axes(projection=ccrs.PlateCarree())
    ax.add_feature(lakes)

    # Get the coordinate reference system used by the data
    transform = ulon.coord_system.as_cartopy_projection()

    # Plot the wind speed as a contour plot
    qplt.contourf(windspeed, 20)

    # Add arrows to show the wind vectors
    plt.quiver(x, y, u, v, pivot="middle", transform=transform)

    plt.title("Wind speed over Lake Victoria")
    qplt.show()

    # Normalise the data for uniform arrow size
    u_norm = u / np.sqrt(u ** 2.0 + v ** 2.0)
    v_norm = v / np.sqrt(u ** 2.0 + v ** 2.0)

    plt.figure()
    ax = plt.axes(projection=ccrs.PlateCarree())
    ax.add_feature(lakes)

    qplt.contourf(windspeed, 20)

    plt.quiver(x, y, u_norm, v_norm, pivot="middle", transform=transform)

    plt.title("Wind speed over Lake Victoria")
    qplt.show()


if __name__ == "__main__":
    main()
```

Total running time of the script: (0 minutes 0.787 seconds)

2.2.5 Deriving Exner Pressure and Air Temperature

This example shows some processing of cubes in order to derive further related cubes; in this case the derived cubes are Exner pressure and air temperature which are calculated by combining air pressure, air potential temperature and specific humidity. Finally, the two new cubes are presented side-by-side in a plot.




```
import matplotlib.pyplot as plt
import matplotlib.ticker

import iris
import iris.coords as coords
import iris.iterate
import iris.plot as iplt
import iris.quickplot as qplt


def limit_colorbar_ticks(contour_object):
    """
    Takes a contour object which has an associated colorbar and limits the
    number of ticks on the colorbar to 4.

    """
    colorbar = contour_object.colorbar
    colorbar.locator = matplotlib.ticker.MaxNLocator(4)
    colorbar.update_ticks()


def main():
    fname = iris.sample_data_path("colpex.pp")

    # The list of phenomena of interest
    phenomena = ["air_potential_temperature", "air_pressure"]
```

(continues on next page)

(continued from previous page)

```
    # Define the constraint on standard name and model level
    constraints = [
        iris.Constraint(phenom, model_level_number=1) for phenom in phenomena
    ]

    air_potential_temperature, air_pressure = iris.load_cubes(
        fname, constraints
    )

    # Define a coordinate which represents 1000 hPa
    p0 = coords.AuxCoord(1000, long_name="P0", units="hPa")
    # Convert reference pressure 'p0' into the same units as 'air_pressure'
    p0.convert_units(air_pressure.units)

    # Calculate Exner pressure
    exner_pressure = (air_pressure / p0) ** (287.05 / 1005.0)
    # Set the name (the unit is scalar)
    exner_pressure.rename("exner_pressure")

    # Calculate air_temp
    air_temperature = exner_pressure * air_potential_temperature
    # Set the name (the unit is K)
    air_temperature.rename("air_temperature")

    # Now create an iterator which will give us lat lon slices of
    # exner pressure and air temperature in the form
    # (exner_slice, air_temp_slice).
    lat_lon_slice_pairs = iris.iterate.izip(
        exner_pressure,
        air_temperature,
        coords=["grid_latitude", "grid_longitude"],
    )

    # For the purposes of this example, we only want to demonstrate the first
    # plot.
    lat_lon_slice_pairs = [next(lat_lon_slice_pairs)]

    plt.figure(figsize=(8, 4))
    for exner_slice, air_temp_slice in lat_lon_slice_pairs:
        plt.subplot(121)
        cont = qplt.contourf(exner_slice)

        # The default colorbar has a few too many ticks on it, causing text to
        # overlap. Therefore, limit the number of ticks.
        limit_colorbar_ticks(cont)

        plt.subplot(122)
        cont = qplt.contourf(air_temp_slice)
        limit_colorbar_ticks(cont)
        iplt.show()
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    main()
```

Total running time of the script: (0 minutes 2.381 seconds)

2.2.6 Global Average Annual Temperature Plot

Produces a time-series plot of North American temperature forecasts for 2 different emission scenarios. Constraining data to a limited spatial area also features in this example.

The data used comes from the HadGEM2-AO model simulations for the A1B and E1 scenarios, both of which were derived using the IMAGE Integrated Assessment Model (Johns et al. 2011; Lowe et al. 2009).

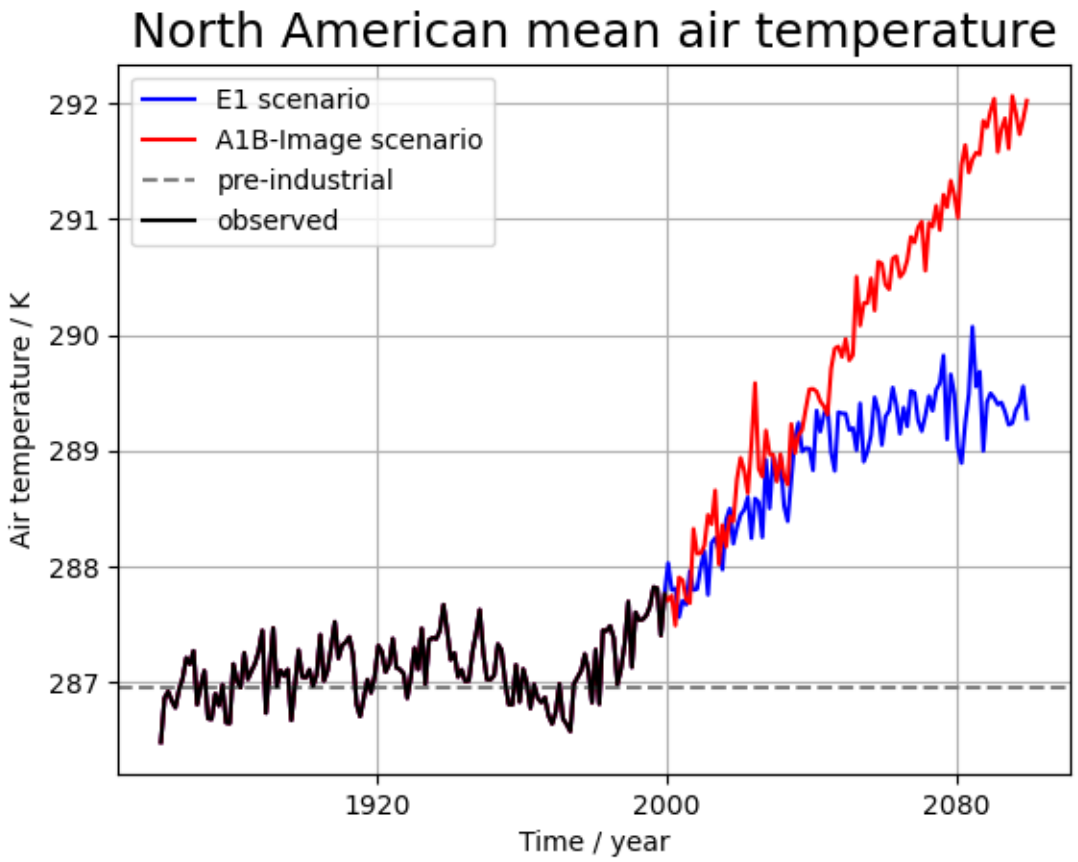
References

Johns T.C., et al. (2011) Climate change under aggressive mitigation: the ENSEMBLES multi-model experiment. Climate Dynamics, Vol 37, No. 9-10, doi:10.1007/s00382-011-1005-5.

Lowe J.A., C.D. Hewitt, D.P. Van Vuuren, T.C. Johns, E. Stehfest, J-F. Royer, and P. van der Linden, 2009. New Study For Climate Modeling, Analyses, and Scenarios. Eos Trans. AGU, Vol 90, No. 21, doi:10.1029/2009EO210001.

See also:

Further details on the aggregation functionality being used in this example can be found in *Cube Statistics*.




```
import matplotlib.pyplot as plt
import numpy as np

import iris
import iris.analysis.cartography
import iris.plot as iplt
import iris.quickplot as qplt


def main():
    # Load data into three Cubes, one for each set of NetCDF files.
    e1 = iris.load_cube(iris.sample_data_path("E1_north_america.nc"))

    a1b = iris.load_cube(iris.sample_data_path("A1B_north_america.nc"))

    # load in the global pre-industrial mean temperature, and limit the domain
    # to the same North American region that e1 and a1b are at.
    north_america = iris.Constraint(
        longitude=lambda v: 225 <= v <= 315, latitude=lambda v: 15 <= v <= 60
    )
    pre_industrial = iris.load_cube(
        iris.sample_data_path("pre-industrial.pp"), north_america
    )
```

(continues on next page)

(continued from previous page)

```
    # Generate area-weights array. As e1 and a1b are on the same grid we can
    # do this just once and re-use. This method requires bounds on lat/lon
    # coords, so let's add some in sensible locations using the "guess_bounds"
    # method.
    e1.coord("latitude").guess_bounds()
    e1.coord("longitude").guess_bounds()
    e1_grid_areas = iris.analysis.cartography.area_weights(e1)
    pre_industrial.coord("latitude").guess_bounds()
    pre_industrial.coord("longitude").guess_bounds()
    pre_grid_areas = iris.analysis.cartography.area_weights(pre_industrial)

    # Perform the area-weighted mean for each of the datasets using the
    # computed grid-box areas.
    pre_industrial_mean = pre_industrial.collapsed(
        ["latitude", "longitude"], iris.analysis.MEAN, weights=pre_grid_areas
    )
    e1_mean = e1.collapsed(
        ["latitude", "longitude"], iris.analysis.MEAN, weights=e1_grid_areas
    )
    a1b_mean = a1b.collapsed(
        ["latitude", "longitude"], iris.analysis.MEAN, weights=e1_grid_areas
    )

    # Plot the datasets
    qplt.plot(e1_mean, label="E1 scenario", lw=1.5, color="blue")
    qplt.plot(a1b_mean, label="A1B-Image scenario", lw=1.5, color="red")

    # Draw a horizontal line showing the pre-industrial mean
    plt.axhline(
        y=pre_industrial_mean.data,
        color="gray",
        linestyle="dashed",
        label="pre-industrial",
        lw=1.5,
    )

    # Constrain the period 1860-1999 and extract the observed data from a1b
    constraint = iris.Constraint(
        time=lambda cell: 1860 <= cell.point.year <= 1999
    )
    observed = a1b_mean.extract(constraint)

    # Assert that this data set is the same as the e1 scenario:
    # they share data up to the 1999 cut off.
    assert np.all(np.isclose(observed.data, e1_mean.extract(constraint).data))

    # Plot the observed data
    qplt.plot(observed, label="observed", color="black", lw=1.5)

    # Add a legend and title
    plt.legend(loc="upper left")
```

(continues on next page)

(continued from previous page)

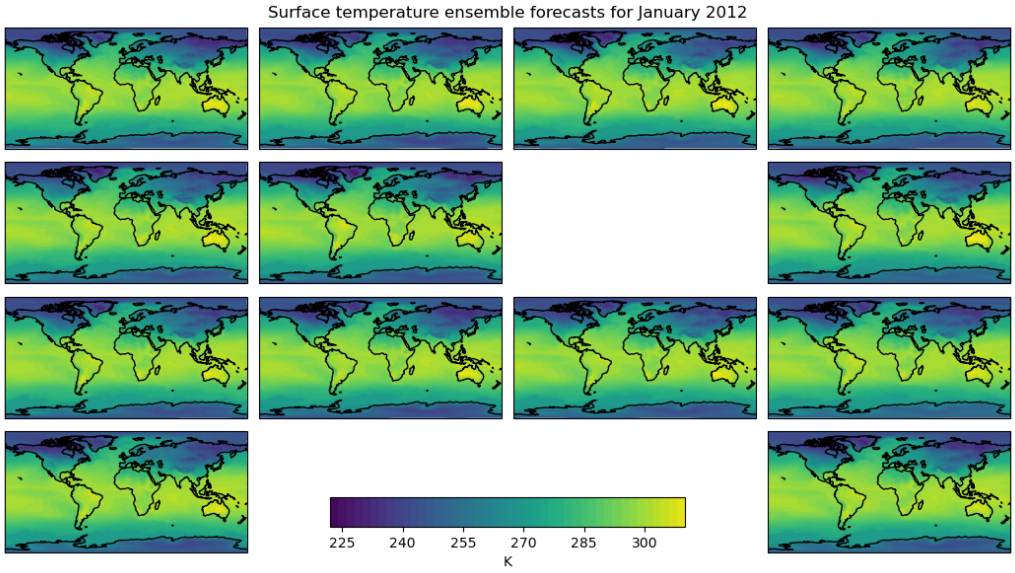
```
    plt.title("North American mean air temperature", fontsize=18)

    plt.xlabel("Time / year")
    plt.grid()
    iplt.show()


if __name__ == "__main__":
    main()
```

Total running time of the script: (0 minutes 0.522 seconds)

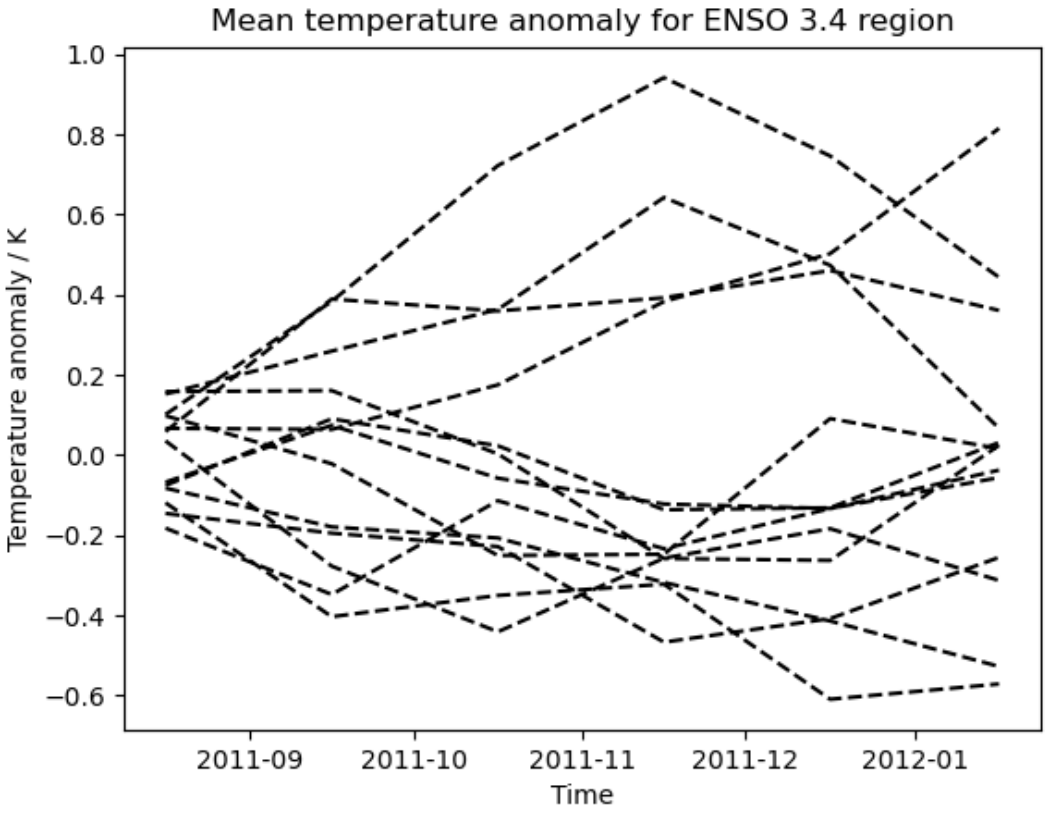
2.2.7 Seasonal Ensemble Model Plots

This example demonstrates the loading of a lagged ensemble dataset from the GloSea4 model, which is then used to produce two types of plot:

- The first shows the "postage stamp" style image with an array of 14 images, one for each ensemble member with a shared colorbar. (The missing image in this example represents ensemble member number 6 which was a failed run)
- The second plot shows the data limited to a region of interest, in this case a region defined for forecasting ENSO (El Nino-Southern Oscillation), which, for the purposes of this example, has had the ensemble mean subtracted from each ensemble member to give an anomaly surface temperature. In practice a better approach would be to take the climatological mean, calibrated to the model, from each ensemble member.



-



-

```
import matplotlib.pyplot as plt
import matplotlib.ticker
import numpy as np

import iris
import iris.plot as iplt


def realization_metadata(cube, field, fname):
    """
    A function which modifies the cube's metadata to add a "realization"
    (ensemble member) coordinate from the filename if one doesn't already exist
    in the cube.

    """
    # Add an ensemble member coordinate if one doesn't already exist.
    if not cube.coords("realization"):
        # The ensemble member is encoded in the filename as *_???.pp where ???
        # is the ensemble member.
        realization_number = fname[-6:-3]
        realization_coord = iris.coords.AuxCoord(
            np.int32(realization_number), "realization", units="1"
        )
        cube.add_aux_coord(realization_coord)
```

(continues on next page)

(continued from previous page)

```
def main():
    # Create a constraint to extract surface temperature cubes which have a
    # "realization" coordinate.
    constraint = iris.Constraint(
        "surface_temperature", realization=lambda value: True
    )
    # Use this to load our ensemble.  The callback ensures all our members
    # have the "realization" coordinate and therefore they will all be loaded.
    surface_temp = iris.load_cube(
        iris.sample_data_path("GloSea4", "ensemble_???.pp"),
        constraint,
        callback=realization_metadata,
    )

    # ---------------------------------------------------------------------
    # Plot #1: Ensemble postage stamps
    # ---------------------------------------------------------------------

    # For the purposes of this example, take the last time element of the cube.
    # First get hold of the last time by slicing the coordinate.
    last_time_coord = surface_temp.coord("time")[-1]
    last_timestep = surface_temp.subset(last_time_coord)

    # Find the maximum and minimum across the dataset.
    data_min = np.min(last_timestep.data)
    data_max = np.max(last_timestep.data)

    # Create a wider than normal figure to support our many plots.
    plt.figure(figsize=(12, 6), dpi=100)

    # Also manually adjust the spacings which are used when creating subplots.
    plt.gcf().subplots_adjust(
        hspace=0.05,
        wspace=0.05,
        top=0.95,
        bottom=0.05,
        left=0.075,
        right=0.925,
    )

    # Iterate over all possible latitude longitude slices.
    for cube in last_timestep.slices(["latitude", "longitude"]):

        # Get the ensemble member number from the ensemble coordinate.
        ens_member = cube.coord("realization").points[0]

        # Plot the data in a 4x4 grid, with each plot's position in the grid
        # being determined by ensemble member number.  The special case for the
        # 13th ensemble member is to have the plot at the bottom right.
        if ens_member == 13:
```

(continues on next page)

(continued from previous page)

```
            plt.subplot(4, 4, 16)
        else:
            plt.subplot(4, 4, ens_member + 1)

        # Plot with 50 evenly spaced contour levels (49 intervals).
        cf = iplt.contourf(cube, 49, vmin=data_min, vmax=data_max)

        # Add coastlines.
        plt.gca().coastlines()

    # Make an axes to put the shared colorbar in.
    colorbar_axes = plt.gcf().add_axes([0.35, 0.1, 0.3, 0.05])
    colorbar = plt.colorbar(cf, colorbar_axes, orientation="horizontal")
    colorbar.set_label(last_timestep.units)

    # Limit the colorbar to 8 tick marks.
    colorbar.locator = matplotlib.ticker.MaxNLocator(8)
    colorbar.update_ticks()

    # Get the time for the entire plot.
    time = last_time_coord.units.num2date(last_time_coord.bounds[0, 0])

    # Set a global title for the postage stamps with the date formated by
    # "monthname year".
    time_string = time.strftime("%B %Y")
    plt.suptitle(f"Surface temperature ensemble forecasts for {time_string}")

    iplt.show()

    # ----------------------------------------------------------------------
    # Plot #2: ENSO plumes
    # ----------------------------------------------------------------------

    # Nino 3.4 lies between: 170W and 120W, 5N and 5S, so use the intersection
    # method to restrict to this region.
    nino_cube = surface_temp.intersection(
        latitude=[-5, 5], longitude=[-170, -120]
    )

    # Calculate the horizontal mean for the nino region.
    mean = nino_cube.collapsed(["latitude", "longitude"], iris.analysis.MEAN)

    # Calculate the ensemble mean of the horizontal mean.
    ensemble_mean = mean.collapsed("realization", iris.analysis.MEAN)

    # Take the ensemble mean from each ensemble member.
    mean -= ensemble_mean

    plt.figure()

    for ensemble_member in mean.slices(["time"]):
        # Draw each ensemble member as a dashed line in black.
```

(continues on next page)

(continued from previous page)

```
        iplt.plot(ensemble_member, "--k")

    plt.title("Mean temperature anomaly for ENSO 3.4 region")
    plt.xlabel("Time")
    plt.ylabel("Temperature anomaly / K")

    iplt.show()


if __name__ == "__main__":
    main()
```

Total running time of the script: (0 minutes 17.347 seconds)

2.2.8 Global Average Annual Temperature Maps

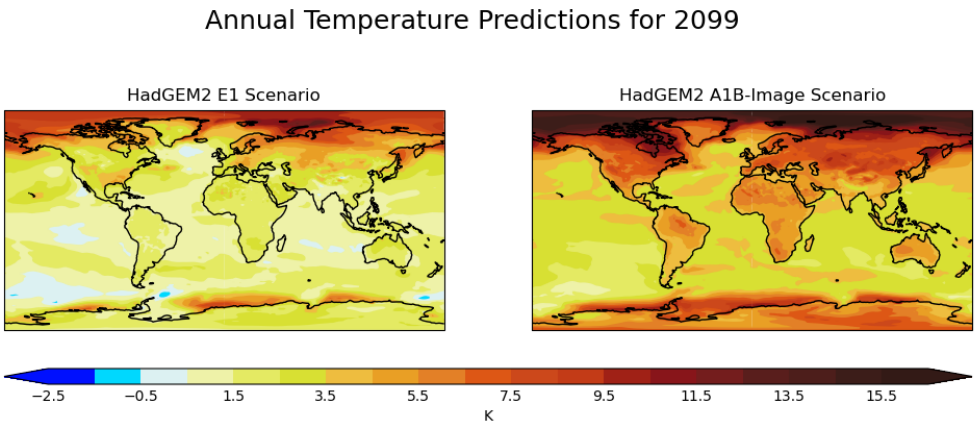
Produces maps of global temperature forecasts from the A1B and E1 scenarios.

The data used comes from the HadGEM2-AO model simulations for the A1B and E1 scenarios, both of which were derived using the IMAGE Integrated Assessment Model (Johns et al. 2011; Lowe et al. 2009).

References

Johns T.C., et al. (2011) Climate change under aggressive mitigation: the ENSEMBLES multi-model experiment. Climate Dynamics, Vol 37, No. 9-10, doi:10.1007/s00382-011-1005-5.

Lowe J.A., C.D. Hewitt, D.P. Van Vuuren, T.C. Johns, E. Stehfest, J-F. Royer, and P. van der Linden, 2009. New Study For Climate Modeling, Analyses, and Scenarios. Eos Trans. AGU, Vol 90, No. 21, doi:10.1029/2009EO210001.




```
import os.path

import matplotlib.pyplot as plt
import numpy as np
```

(continues on next page)

(continued from previous page)

```
import iris
import iris.coords as coords
import iris.plot as iplt


def cop_metadata_callback(cube, field, filename):
    """
    A function which adds an "Experiment" coordinate which comes from the
    filename.
    """

    # Extract the experiment name (such as A1B or E1) from the filename (in
    # this case it is just the start of the file name, before the first ".").
    fname = os.path.basename(filename)  # filename without path.
    experiment_label = fname.split(".")[0]

    # Create a coordinate with the experiment label in it...
    exp_coord = coords.AuxCoord(
        experiment_label, long_name="Experiment", units="no_unit"
    )

    # ...and add it to the cube.
    cube.add_aux_coord(exp_coord)


def main():
    # Load E1 and A1B scenarios using the callback to update the metadata.
    scenario_files = [
        iris.sample_data_path(fname) for fname in ["E1.2098.pp", "A1B.2098.pp"]
    ]
    scenarios = iris.load(scenario_files, callback=cop_metadata_callback)

    # Load the preindustrial reference data.
    preindustrial = iris.load_cube(iris.sample_data_path("pre-industrial.pp"))

    # Define evenly spaced contour levels: -2.5, -1.5, ... 15.5, 16.5 with the
    # specific colours.
    levels = np.arange(20) - 2.5
    red = (
        np.array(
            [
                0,
                0,
                221,
                239,
                229,
                217,
                239,
                234,
                228,
                222,
                205,
```

(continues on next page)

(continued from previous page)

```
                196,
                161,
                137,
                116,
                89,
                77,
                60,
                51,
            ]
        )
        / 256.0
    )
    green = (
        np.array(
            [
                16,
                217,
                242,
                243,
                235,
                225,
                190,
                160,
                128,
                87,
                72,
                59,
                33,
                21,
                29,
                30,
                30,
                29,
                26,
            ]
        )
        / 256.0
    )
    blue = (
        np.array(
            [
                255,
                255,
                243,
                169,
                99,
                51,
                63,
                37,
                39,
                21,
                27,
```

(continues on next page)

(continued from previous page)

```
                23,
                22,
                26,
                29,
                28,
                27,
                25,
                22,
            ]
        )
        / 256.0
    )

    # Put those colours into an array which can be passed to contourf as the
    # specific colours for each level.
    colors = np.stack([red, green, blue], axis=1)

    # Make a wider than normal figure to house two maps side-by-side.
    fig, ax_array = plt.subplots(1, 2, figsize=(12, 5))

    # Loop over our scenarios to make a plot for each.
    for ax, experiment, label in zip(
        ax_array, ["E1", "A1B"], ["E1", "A1B-Image"]
    ):
        exp_cube = scenarios.extract_cube(
            iris.Constraint(Experiment=experiment)
        )
        time_coord = exp_cube.coord("time")

        # Calculate the difference from the preindustial control run.
        exp_anom_cube = exp_cube - preindustrial

        # Plot this anomaly.
        plt.sca(ax)
        ax.set_title(f"HadGEM2 {label} Scenario", fontsize=10)
        contour_result = iplt.contourf(
            exp_anom_cube, levels, colors=colors, extend="both"
        )
        plt.gca().coastlines()

    # Now add a colourbar who's leftmost point is the same as the leftmost
    # point of the left hand plot and rightmost point is the rightmost
    # point of the right hand plot.

    # Get the positions of the 2nd plot and the left position of the 1st plot.
    left, bottom, width, height = ax_array[1].get_position().bounds
    first_plot_left = ax_array[0].get_position().bounds[0]

    # The width of the colorbar should now be simple.
    width = left - first_plot_left + width

    # Add axes to the figure, to place the colour bar.
```

(continues on next page)

(continued from previous page)

```
    colorbar_axes = fig.add_axes([first_plot_left, 0.18, width, 0.03])

    # Add the colour bar.
    cbar = plt.colorbar(
        contour_result, colorbar_axes, orientation="horizontal"
    )

    # Label the colour bar and add ticks.
    cbar.set_label(preindustrial.units)
    cbar.ax.tick_params(length=0)

    # Get the time datetime from the coordinate.
    time = time_coord.units.num2date(time_coord.points[0])
    # Set a title for the entire figure, using the year from the datetime
    # object. Also, set the y value for the title so that it is not tight to
    # the top of the plot.
    fig.suptitle(
        f"Annual Temperature Predictions for {time.year}",
        y=0.9,
        fontsize=18,
    )

    iplt.show()


if __name__ == "__main__":
    main()
```

Total running time of the script: (0 minutes 1.399 seconds)

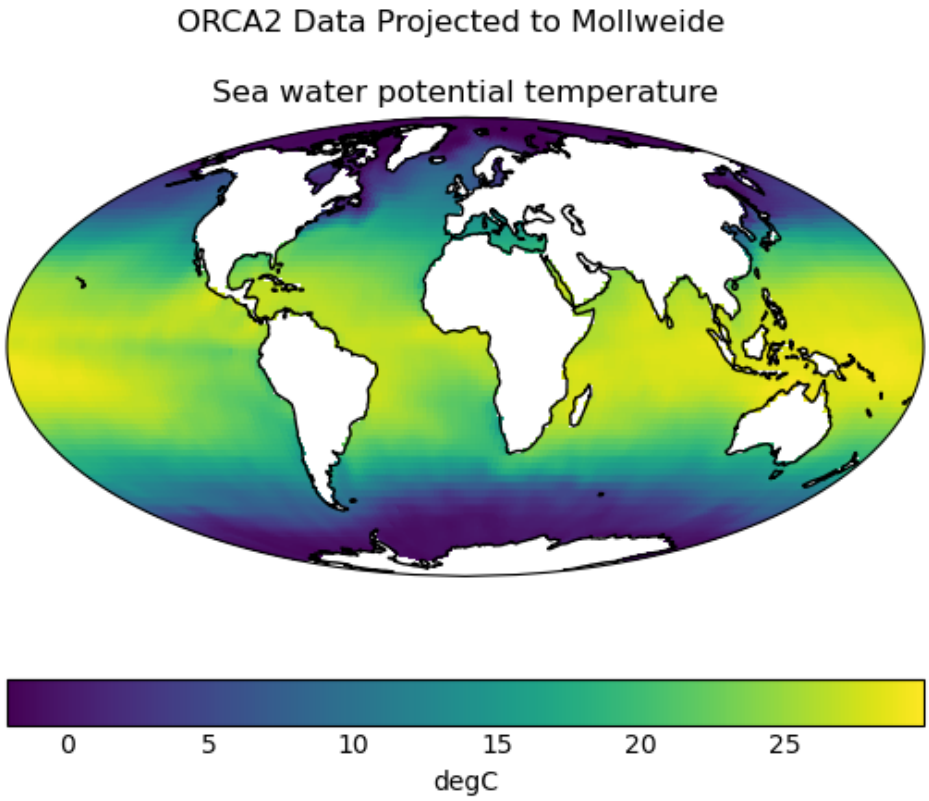
2.3 Oceanography

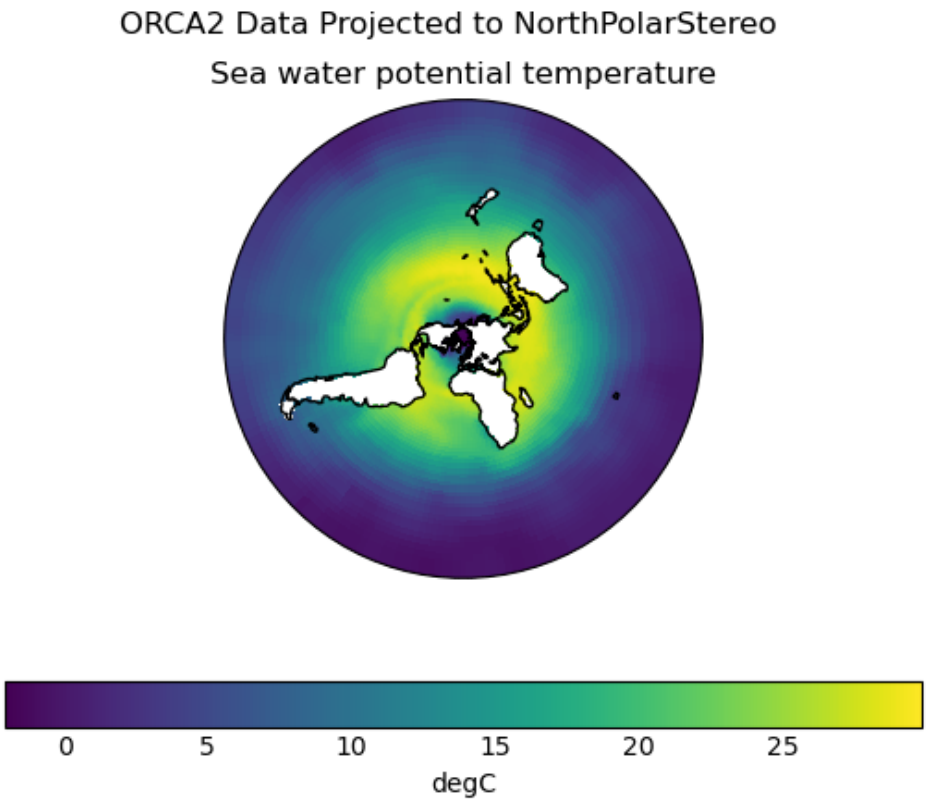
2.3.1 Tri-Polar Grid Projected Plotting

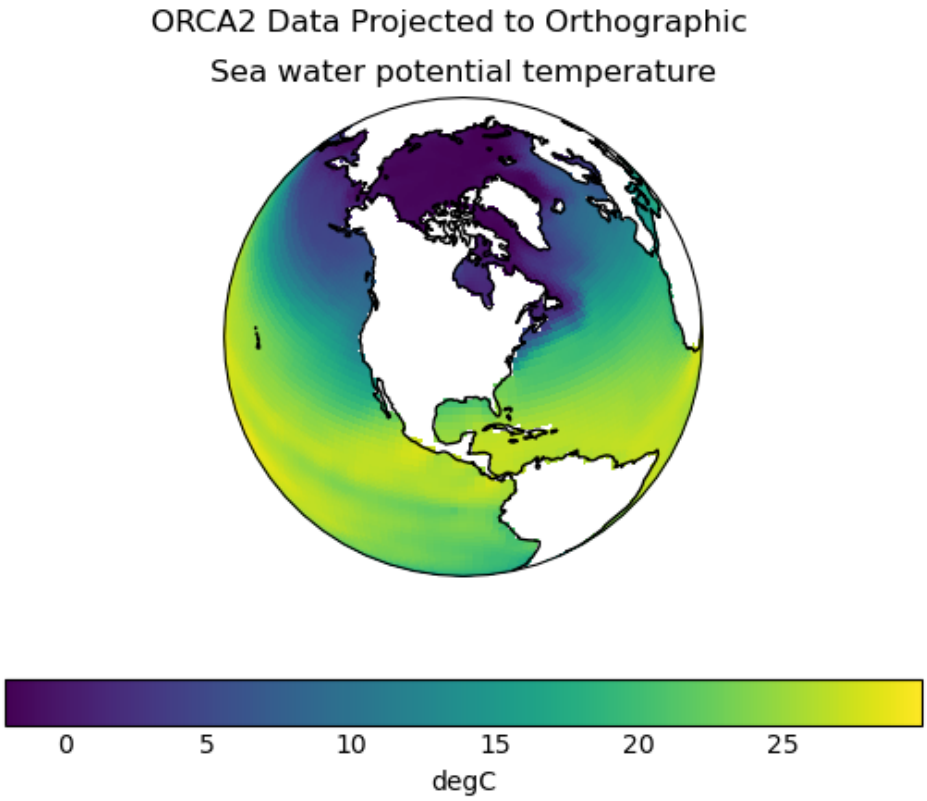
This example demonstrates cell plots of data on the semi-structured ORCA2 model grid.

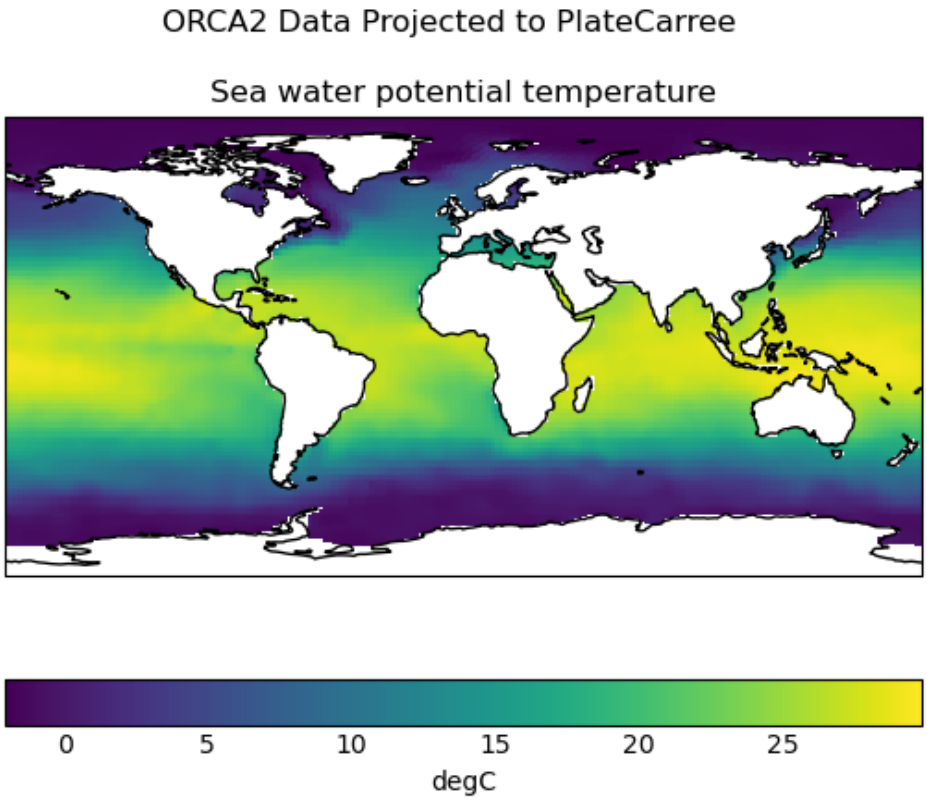
First, the data is projected into the PlateCarree coordinate reference system.

Second four pcolormesh plots are created from this projected dataset, using different projections for the output image.


ORCA2 Data Projected to Mollweide
Sea water potential temperature
0
5
10
15
20
25
degC


ORCA2 Data Projected to NorthPolarStereo
Sea water potential temperature
0
5
10
15
20
25
degC


ORCA2 Data Projected to Orthographic
Sea water potential temperature
0
5
10
15
20
25
degC



-

```
import cartopy.crs as ccrs
import matplotlib.pyplot as plt

import iris
import iris.analysis.cartography
import iris.plot as iplt
import iris.quickplot as qplt


def main():
    # Load data
    filepath = iris.sample_data_path("orca2_votemper.nc")
    cube = iris.load_cube(filepath)

    # Choose plot projections
    projections = {}
    projections["Mollweide"] = ccrs.Mollweide()
    projections["PlateCarree"] = ccrs.PlateCarree()
    projections["NorthPolarStereo"] = ccrs.NorthPolarStereo()
    projections["Orthographic"] = ccrs.Orthographic(
        central_longitude=-90, central_latitude=45
    )

    pcarree = projections["PlateCarree"]
```

(continues on next page)

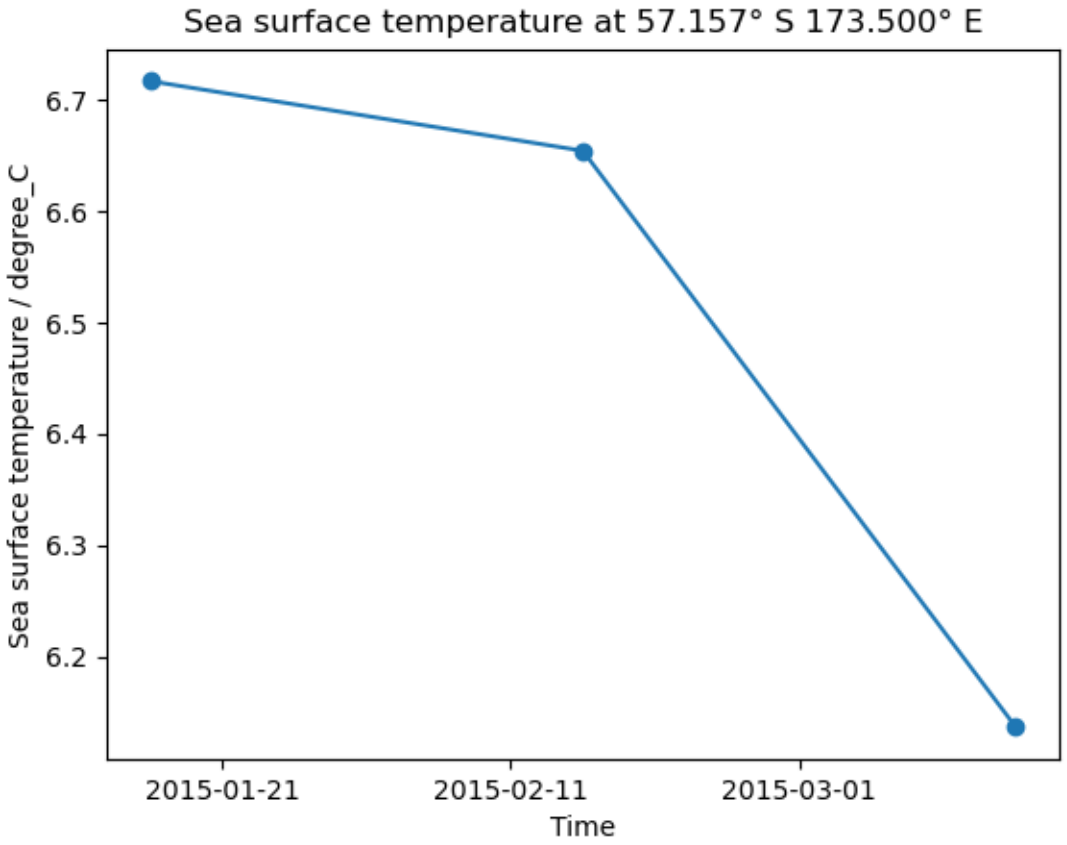
(continued from previous page)

```
    # Transform cube to target projection
    new_cube, extent = iris.analysis.cartography.project(
        cube, pcarree, nx=400, ny=200
    )

    # Plot data in each projection
    for name in sorted(projections):
        fig = plt.figure()
        fig.suptitle("ORCA2 Data Projected to {}".format(name))
        # Set up axes and title
        ax = plt.subplot(projection=projections[name])
        # Set limits
        ax.set_global()
        # plot with Iris quickplot pcolormesh
        qplt.pcolormesh(new_cube)
        # Draw coastlines
        ax.coastlines()

        iplt.show()


if __name__ == "__main__":
    main()
```

Total running time of the script: (0 minutes 3.455 seconds)

2.3.2 Load a Time Series of Data From the NEMO Model

This example demonstrates how to load multiple files containing data output by the NEMO model and combine them into a time series in a single cube. The different time dimensions in these files can prevent Iris from concatenating them without the intervention shown here.




```
import matplotlib.pyplot as plt

import iris
import iris.plot as iplt
import iris.quickplot as qplt
from iris.util import promote_aux_coord_to_dim_coord


def main():
    # Load the three files of sample NEMO data.
    fname = iris.sample_data_path("NEMO/nemo_1m_*.nc")
    cubes = iris.load(fname)

    # Some attributes are unique to each file and must be blanked
    # to allow concatenation.
    differing_attrs = ["file_name", "name", "timeStamp", "TimeStamp"]
    for cube in cubes:
        for attribute in differing_attrs:
            cube.attributes[attribute] = ""

    # The cubes still cannot be concatenated because their time dimension is
    # time_counter rather than time. time needs to be promoted to allow
    # concatenation.
```

(continues on next page)

(continued from previous page)

```
    for cube in cubes:
        promote_aux_coord_to_dim_coord(cube, "time")

    # The cubes can now be concatenated into a single time series.
    cube = cubes.concatenate_cube()

    # Generate a time series plot of a single point
    plt.figure()
    y_point_index = 100
    x_point_index = 100
    qplt.plot(cube[:, y_point_index, x_point_index], "o-")

    # Include the point's position in the plot's title
    lat_point = cube.coord("latitude").points[y_point_index, x_point_index]
    lat_string = "{:.3f}\u00B0 {}".format(
        abs(lat_point), "N" if lat_point > 0.0 else "S"
    )
    lon_point = cube.coord("longitude").points[y_point_index, x_point_index]
    lon_string = "{:.3f}\u00B0 {}".format(
        abs(lon_point), "E" if lon_point > 0.0 else "W"
    )
    plt.title(
        "{} at {} {}".format(
            cube.long_name.capitalize(), lat_string, lon_string
        )
    )

    iplt.show()


if __name__ == "__main__":
    main()
```

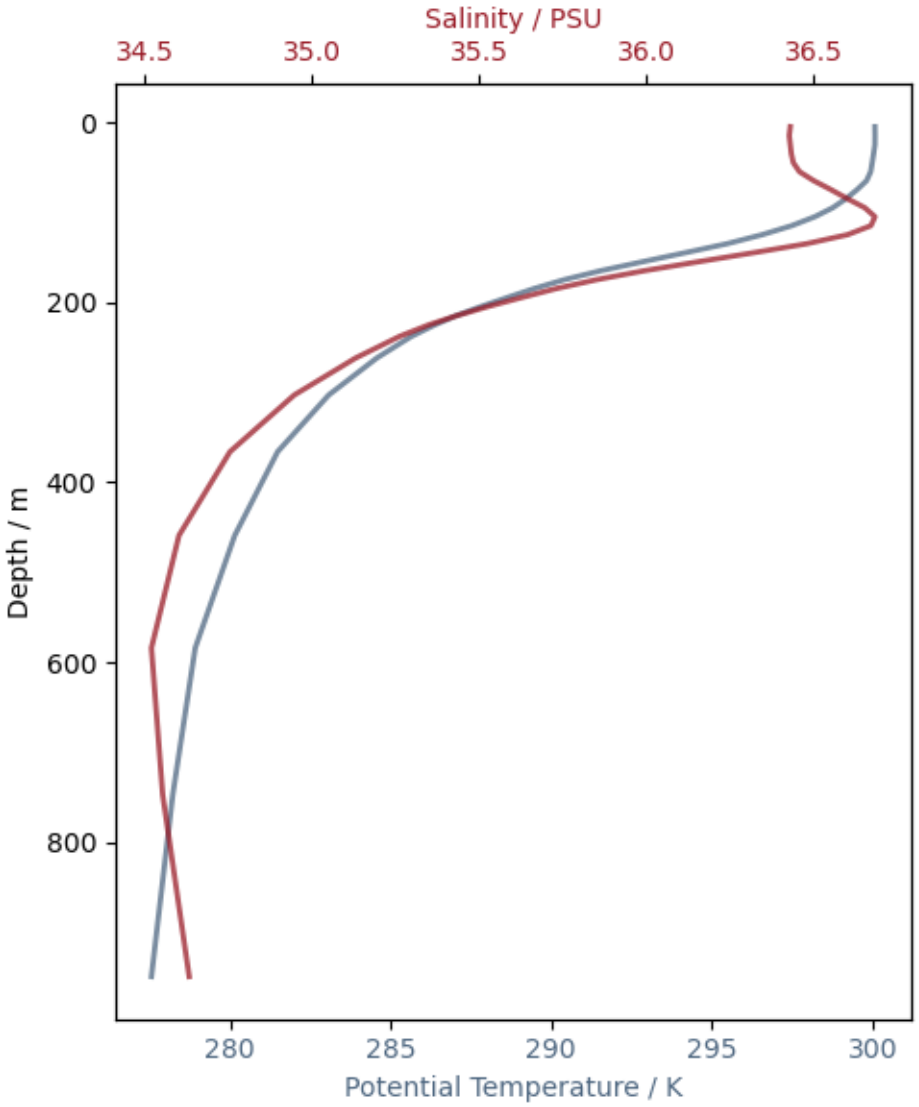
Total running time of the script: (0 minutes 0.286 seconds)

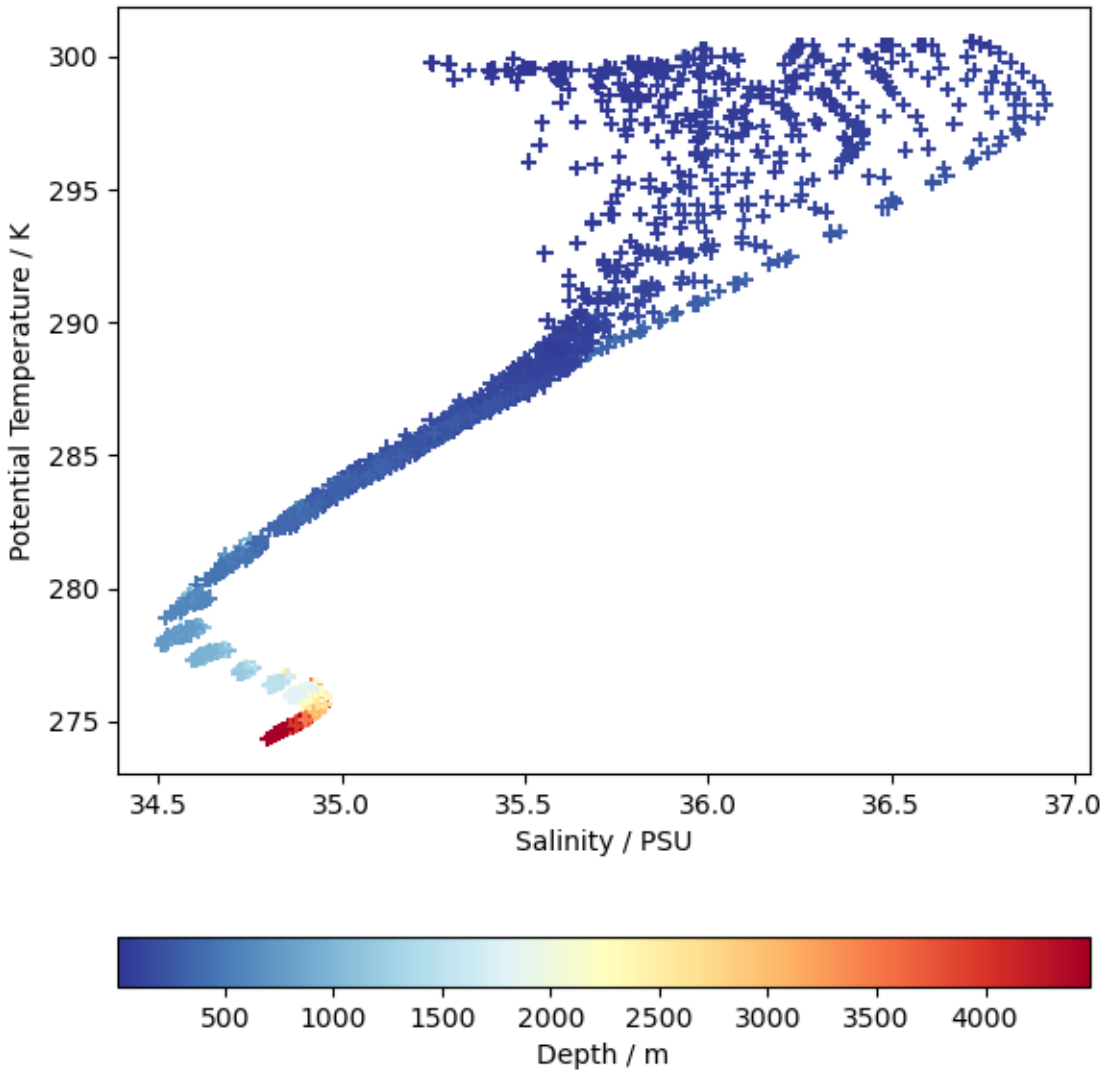
2.3.3 Oceanographic Profiles and T-S Diagrams

This example demonstrates how to plot vertical profiles of different variables in the same axes, and how to make a scatter plot of two variables. There is an oceanographic theme but the same techniques are equally applicable to atmospheric or other kinds of data.

The data used are profiles of potential temperature and salinity in the Equatorial and South Atlantic, output from an ocean model.

The y-axis of the first plot produced will be automatically inverted due to the presence of the attribute positive=down on the depth coordinate. This means depth values intuitively increase downward on the y-axis.


Salinity / PSU
34.5
35.0
35.5
36.0
36.5
0
200
400
600
800
Depth / m
280
285
290
295
300
Potential Temperature / K



-

```
import matplotlib.pyplot as plt

import iris
import iris.iterate
import iris.plot as iplt


def main():
    # Load the gridded temperature and salinity data.
    fname = iris.sample_data_path("atlantic_profiles.nc")
    cubes = iris.load(fname)
    (theta,) = cubes.extract("sea_water_potential_temperature")
    (salinity,) = cubes.extract("sea_water_practical_salinity")

    # Extract profiles of temperature and salinity from a particular point in
    # the southern portion of the domain, and limit the depth of the profile
```

(continues on next page)

(continued from previous page)

```
    # to 1000m.
    lon_cons = iris.Constraint(longitude=330.5)
    lat_cons = iris.Constraint(latitude=lambda l: -10 < l < -9)
    depth_cons = iris.Constraint(depth=lambda d: d <= 1000)
    theta_1000m = theta.extract(depth_cons & lon_cons & lat_cons)
    salinity_1000m = salinity.extract(depth_cons & lon_cons & lat_cons)

    # Plot these profiles on the same set of axes. Depth is automatically
    # recognised as a vertical coordinate and placed on the y-axis.
    # The first plot is in the default axes. We'll use the same color for the
    # curve and its axes/tick labels.
    plt.figure(figsize=(5, 6))
    temperature_color = (0.3, 0.4, 0.5)
    ax1 = plt.gca()
    iplt.plot(
        theta_1000m,
        linewidth=2,
        color=temperature_color,
        alpha=0.75,
    )
    ax1.set_xlabel("Potential Temperature / K", color=temperature_color)
    ax1.set_ylabel("Depth / m")
    for ticklabel in ax1.get_xticklabels():
        ticklabel.set_color(temperature_color)

    # To plot salinity in the same axes we use twiny(). We'll use a different
    # color to identify salinity.
    salinity_color = (0.6, 0.1, 0.15)
    ax2 = plt.gca().twiny()
    iplt.plot(
        salinity_1000m,
        linewidth=2,
        color=salinity_color,
        alpha=0.75,
    )
    ax2.set_xlabel("Salinity / PSU", color=salinity_color)
    for ticklabel in ax2.get_xticklabels():
        ticklabel.set_color(salinity_color)
    plt.tight_layout()
    iplt.show()

    # Now plot a T-S diagram using scatter. We'll use all the profiles here,
    # and each point will be coloured according to its depth.
    plt.figure(figsize=(6, 6))
    depth_values = theta.coord("depth").points
    for s, t in iris.iterate.izip(salinity, theta, coords="depth"):
        iplt.scatter(s, t, c=depth_values, marker="+", cmap="RdYlBu_r")
    ax = plt.gca()
    ax.set_xlabel("Salinity / PSU")
    ax.set_ylabel("Potential Temperature / K")
    cb = plt.colorbar(orientation="horizontal")
    cb.set_label("Depth / m")
```

(continues on next page)

(continued from previous page)

```
    plt.tight_layout()
    iplt.show()


if __name__ == "__main__":
    main()
```

Total running time of the script: (0 minutes 1.154 seconds)

CHAPTER

THREE

INTRODUCTION

If you are reading this user guide for the first time it is strongly recommended that you read the user guide fully before experimenting with your own data files.

Much of the content has supplementary links to the reference documentation; you will not need to follow these links in order to understand the guide but they may serve as a useful reference for future exploration.

- *Iris Data Structures*
- *Loading Iris Cubes*
- *Saving Iris Cubes*
- *Navigating a Cube*
- *Subsetting a Cube*
- *Real and Lazy Data*
- *Plotting a Cube*
- *Cube Interpolation and Regridding*
- *Merge and Concatenate*
- *Cube Statistics*
- *Cube Maths*
- *Citing Iris*
- *Code Maintenance*

CHAPTER

FOUR

IRIS DATA STRUCTURES

The top level object in Iris is called a cube. A cube contains data and metadata about a phenomenon.

In Iris, a cube is an interpretation of the *Climate and Forecast (CF) Metadata Conventions* whose purpose is to:

> *require conforming datasets to contain sufficient metadata that they are self-describing… including physical units if appropriate, and that each value can be located in space (relative to earth-based coordinates) and time.*

Whilst the CF conventions are often mentioned alongside NetCDF, Iris implements several major format importers which can take files of specific formats and turn them into Iris cubes. Additionally, a framework is provided which allows users to extend Iris' import capability to cater for specialist or unimplemented formats.

A single cube describes one and only one phenomenon, always has a name, a unit and an n-dimensional data array to represents the cube's phenomenon. In order to locate the data spatially, temporally, or in any other higher-dimensional space, a collection of *coordinates* exist on the cube.

4.1 Coordinates

A coordinate is a container to store metadata about some dimension(s) of a cube's data array and therefore, by definition, its phenomenon.

- Each coordinate has a name and a unit.
- When a coordinate is added to a cube, the data dimensions that it represents are also provided.
  - The shape of a coordinate is always the same as the shape of the associated data dimension(s) on the cube.
  - A dimension not explicitly listed signifies that the coordinate is independent of that dimension.
  - Each dimension of a coordinate must be mapped to a data dimension. The only coordinates with no mapping are scalar coordinates.
- Depending on the underlying data that the coordinate is representing, its values may be discrete points or be bounded to represent interval extents (e.g. temperature at *point x* **vs** rainfall accumulation *between 0000-1200 hours*).
- Coordinates have an attributes dictionary which can hold arbitrary extra metadata, excluding certain restricted CF names
- More complex coordinates may contain a coordinate system which is necessary to fully interpret the values contained within the coordinate.

There are two classes of coordinates:

DimCoord

- Numeric

- Monotonic
- Representative of, at most, a single data dimension (1d)

AuxCoord

- May be of any type, including strings
- May represent multiple data dimensions (n-dimensional)

4.2 Cube

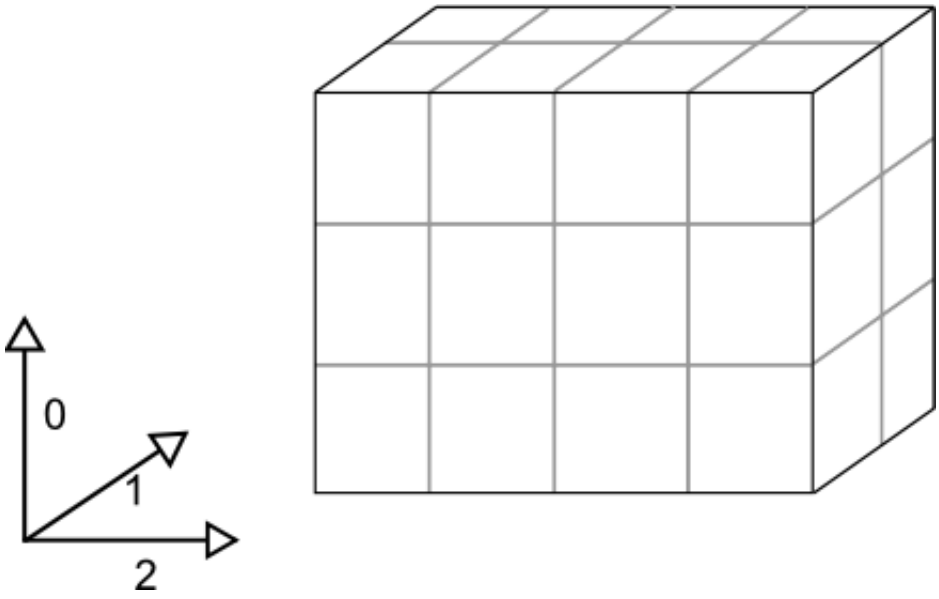
A cube consists of:

- a standard name and/or a long name and an appropriate unit
- a data array who's values are representative of the phenomenon
- a collection of coordinates and associated data dimensions on the cube's data array, which are split into two separate lists:
  - *dimension coordinates* - DimCoords which uniquely map to exactly one data dimension, ordered by dimension.
  - *auxiliary coordinates* - DimCoords or AuxCoords which map to as many data dimensions as the coordinate has dimensions.
- an attributes dictionary which, other than some protected CF names, can hold arbitrary extra metadata.
- a list of cell methods to represent operations which have already been applied to the data (e.g. "mean over time")
- a list of coordinate "factories" used for deriving coordinates from the values of other coordinates in the cube

4.2.1 Cubes in Practice

4.3 A Simple Cube Example

Suppose we have some gridded data which has 24 air temperature readings (in Kelvin) which is located at 4 different longitudes, 2 different latitudes and 3 different heights. Our data array can be represented pictorially:

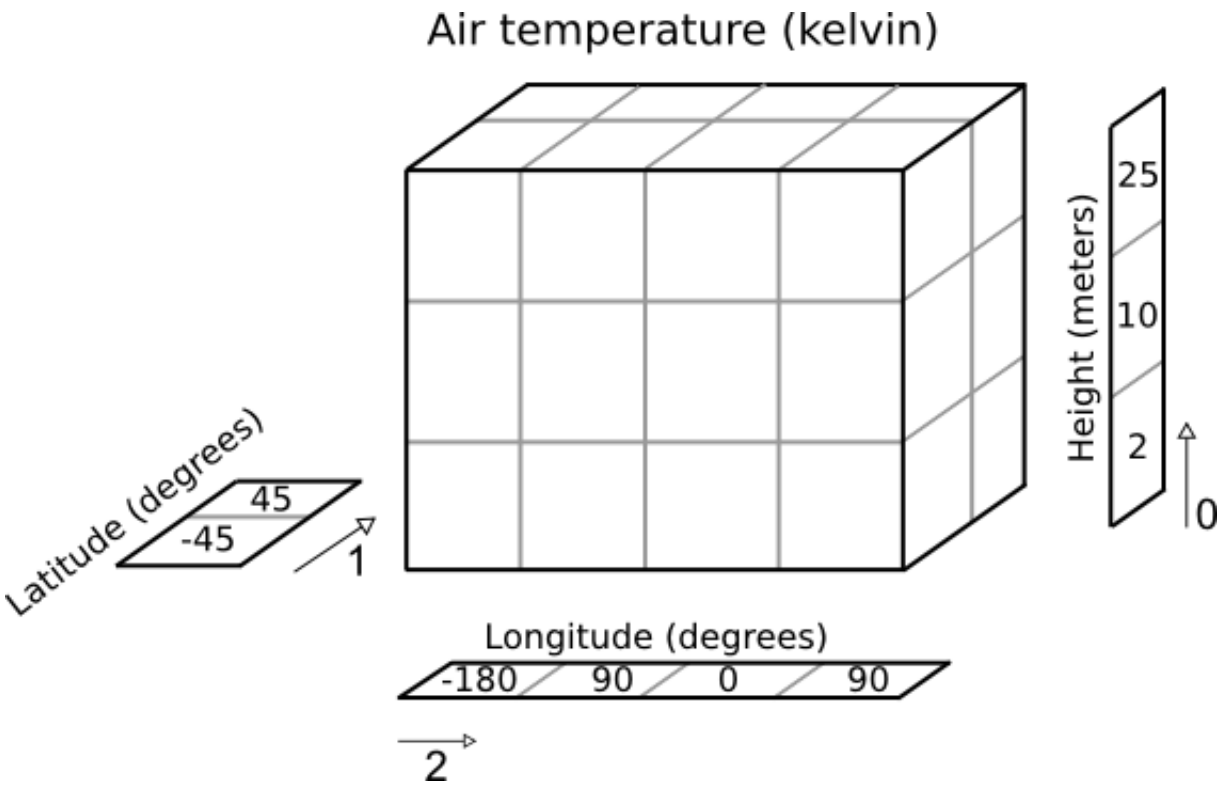


Where dimensions 0, 1, and 2 have lengths 3, 2 and 4 respectively.

The Iris cube to represent this data would consist of:

- a standard name of `air_temperature` and a unit of `kelvin`
- a data array of shape `(3, 2, 4)`
- a coordinate, mapping to dimension 0, consisting of:
  - a standard name of `height` and unit of `meters`
  - an array of length 3 representing the 3 `height` points
- a coordinate, mapping to dimension 1, consisting of:
  - a standard name of `latitude` and unit of `degrees`
  - an array of length 2 representing the 2 latitude points
  - a coordinate system such that the `latitude` points could be fully located on the globe
- a coordinate, mapping to dimension 2, consisting of:
  - a standard name of `longitude` and unit of `degrees`
  - an array of length 4 representing the 4 longitude points
  - a coordinate system such that the `longitude` points could be fully located on the globe

Pictorially the cube has taken on more information than a simple array:



Additionally further information may be optionally attached to the cube. For example, it is possible to attach any of the following:

- a coordinate, not mapping to any data dimensions, consisting of:
  - a standard name of `time` and unit of `days since 2000-01-01 00:00`
  - a data array of length 1 representing the time that the data array is valid for
- an auxiliary coordinate, mapping to dimensions 1 and 2, consisting of:
  - a long name of `place name` and no unit
  - a 2d string array of shape `(2, 4)` with the names of the 8 places that the lat/lons correspond to
- an auxiliary coordinate "factory", which can derive its own mapping, consisting of:
  - a standard name of `height` and a unit of `feet`
  - knowledge of how data values for this coordinate can be calculated given the `height in meters` coordinate
- a cell method of "mean" over "ensemble" to indicate that the data has been meaned over a collection of "ensembles" (i.e. multiple model runs).

4.4 Printing a Cube

Every Iris cube can be printed to screen as you will see later in the user guide. It is worth familiarising yourself with the output as this is the quickest way of inspecting the contents of a cube. Here is the result of printing a real life cube:

```
air_potential_temperature / (K)     (time: 3; model_level_number: 7; grid_latitude: 204;␣
↪grid_longitude: 187)
    Dimension coordinates:
        time                             x                      -                 -      ␣
↪              -
        model_level_number               -                      x                 -      ␣
↪              -
        grid_latitude                    -                      -                 x      ␣
↪              -
        grid_longitude                   -                      -                 -      ␣
↪              x
    Auxiliary coordinates:
        forecast_period                  x                      -                 -      ␣
↪              -
        level_height                     -                      x                 -      ␣
↪              -
        sigma                            -                      x                 -      ␣
↪              -
        surface_altitude                 -                      -                 x      ␣
↪              x
    Derived coordinates:
        altitude                         -                      x                 x      ␣
↪              x
    Scalar coordinates:
        forecast_reference_time     2009-11-19 04:00:00
    Attributes:
        STASH                       m01s00i004
        source                      Data from Met Office Unified Model
        um_version                  7.3
```

Using this output we can deduce that:

- The cube represents air potential temperature.
- There are 4 data dimensions, and the data has a shape of `(3, 7, 204, 187)`
- The 4 data dimensions are mapped to the `time`, `model_level_number`, `grid_latitude`, `grid_longitude` coordinates respectively
- There are three 1d auxiliary coordinates and one 2d auxiliary (`surface_altitude`)
- There is a single `altitude` derived coordinate, which spans 3 data dimensions
- There are 7 distinct values in the "model_level_number" coordinate. Similar inferences can be made for the other dimension coordinates.
- There are 7, not necessarily distinct, values in the `level_height` coordinate.
- There is a single `forecast_reference_time` scalar coordinate representing the entire cube.
- The cube has one further attribute relating to the phenomenon. In this case the originating file format, PP, encodes information in a STASH code which in some cases can be useful for identifying advanced experiment information relating to the phenomenon.

CHAPTER

FIVE

LOADING IRIS CUBES

To load a single file into a **list** of Iris cubes the *iris.load()* function is used:

```
import iris
filename = '/path/to/file'
cubes = iris.load(filename)
```

Iris will attempt to return **as few cubes as possible** by collecting together multiple fields with a shared standard name into a single multidimensional cube.

The *iris.load()* function automatically recognises the format of the given files and attempts to produce Iris Cubes from their contents.

Note: Currently there is support for CF NetCDF, GRIB 1 & 2, PP and FieldsFiles file formats with a framework for this to be extended to custom formats.

In order to find out what has been loaded, the result can be printed:

```
>>> import iris
>>> filename = iris.sample_data_path('uk_hires.pp')
>>> cubes = iris.load(filename)
>>> print(cubes)
0: air_potential_temperature / (K)     (time: 3; model_level_number: 7; grid_latitude:␣
↪204; grid_longitude: 187)
1: surface_altitude / (m)              (grid_latitude: 204; grid_longitude: 187)
```

This shows that there were 2 cubes as a result of loading the file, they were: `air_potential_temperature` and `surface_altitude`.

The `surface_altitude` cube was 2 dimensional with:

- the two dimensions have extents of 204 and 187 respectively and are represented by the `grid_latitude` and `grid_longitude` coordinates.

The `air_potential_temperature` cubes were 4 dimensional with:

- the same length `grid_latitude` and `grid_longitude` dimensions as `surface_altitide`
- a `time` dimension of length 3
- a `model_level_number` dimension of length 7

Note: The result of *iris.load()* is **always** a *list of cubes*. Anything that can be done with a Python `list` can be done with the resultant list of cubes. It is worth noting, however, that there is no inherent order to this *list*

of cubes. Because of this, indexing may be inconsistent. A more consistent way to extract a cube is by using the *iris.Constraint* class as described in *Constrained Loading*.

Hint: Throughout this user guide you will see the function `iris.sample_data_path` being used to get the filename for the resources used in the examples. The result of this function is just a string.

Using this function allows us to provide examples which will work across platforms and with data installed in different locations, however in practice you will want to use your own strings:

```
filename = '/path/to/file'
cubes = iris.load(filename)
```

To get the air potential temperature cube from the list of cubes returned by *iris.load()* in the previous example, list indexing can be used:

```
>>> import iris
>>> filename = iris.sample_data_path('uk_hires.pp')
>>> cubes = iris.load(filename)
>>> # get the first cube (list indexing is 0 based)
>>> air_potential_temperature = cubes[0]
>>> print(air_potential_temperature)
air_potential_temperature / (K)     (time: 3; model_level_number: 7; grid_latitude: 204;␣
↪grid_longitude: 187)
    Dimension coordinates:
        time                             x                      -                      -      ␣
↪              -
        model_level_number               -                      x                      -      ␣
↪              -
        grid_latitude                    -                      -                      x      ␣
↪              -
        grid_longitude                   -                      -                      -      ␣
↪              x
    Auxiliary coordinates:
        forecast_period                  x                      -                      -      ␣
↪              -
        level_height                     -                      x                      -      ␣
↪              -
        sigma                            -                      x                      -      ␣
↪              -
        surface_altitude                 -                      -                      x      ␣
↪              x
    Derived coordinates:
        altitude                         -                      x                      x      ␣
↪              x
    Scalar coordinates:
        forecast_reference_time     2009-11-19 04:00:00
    Attributes:
        STASH                       m01s00i004
        source                      Data from Met Office Unified Model
        um_version                  7.3
```

Notice that the result of printing a **cube** is a little more verbose than it was when printing a **list of cubes**. In addition

to the very short summary which is provided when printing a list of cubes, information is provided on the coordinates which constitute the cube in question. This was the output discussed at the end of the *Iris Data Structures* section.

Note: Dimensioned coordinates will have a dimension marker `x` in the appropriate column for each cube data dimension that they describe.

5.1 Loading Multiple Files

To load more than one file into a list of cubes, a list of filenames can be provided to *iris.load()*:

```
filenames = [iris.sample_data_path('uk_hires.pp'),
             iris.sample_data_path('air_temp.pp')]
cubes = iris.load(filenames)
```

It is also possible to load one or more files with wildcard substitution using the expansion rules defined `fnmatch`.

For example, to match **zero or more characters** in the filename, star wildcards can be used:

```
filename = iris.sample_data_path('GloSea4', '*.pp')
cubes = iris.load(filename)
```

Note: The cubes returned will not necessarily be in the same order as the order of the filenames.

5.2 Lazy Loading

In fact when Iris loads data from most file types, it normally only reads the essential descriptive information or metadata : the bulk of the actual data content will only be loaded later, as it is needed. This is referred to as 'lazy' data. It allows loading to be much quicker, and to occupy less memory.

For more on the benefits, handling and uses of lazy data, see *Real and Lazy Data*.

5.3 Constrained Loading

Given a large dataset, it is possible to restrict or constrain the load to match specific Iris cube metadata. Constrained loading provides the ability to generate a cube from a specific subset of data that is of particular interest.

As we have seen, loading the following file creates several Cubes:

```
filename = iris.sample_data_path('uk_hires.pp')
cubes = iris.load(filename)
```

Specifying a name as a constraint argument to *iris.load()* will mean only cubes with matching `name` will be returned:

```
filename = iris.sample_data_path('uk_hires.pp')
cubes = iris.load(filename, 'surface_altitude')
```

Note that, the provided name will match against either the standard name, long name, NetCDF variable name or STASH metadata of a cube. Therefore, the previous example using the `surface_altitude` standard name constraint can also be achieved using the STASH value of `m01s00i033`:

```
filename = iris.sample_data_path('uk_hires.pp')
cubes = iris.load(filename, 'm01s00i033')
```

If further specific name constraint control is required i.e., to constrain against a combination of standard name, long name, NetCDF variable name and/or STASH metadata, consider using the *iris.NameConstraint*. For example, to constrain against both a standard name of `surface_altitude` **and** a STASH of `m01s00i033`:

```
filename = iris.sample_data_path('uk_hires.pp')
constraint = iris.NameConstraint(standard_name='surface_altitude', STASH='m01s00i033')
cubes = iris.load(filename, constraint)
```

To constrain the load to multiple distinct constraints, a list of constraints can be provided. This is equivalent to running load once for each constraint but is likely to be more efficient:

```
filename = iris.sample_data_path('uk_hires.pp')
cubes = iris.load(filename, ['air_potential_temperature', 'surface_altitude'])
```

The *iris.Constraint* class can be used to restrict coordinate values on load. For example, to constrain the load to match a specific `model_level_number`:

```
filename = iris.sample_data_path('uk_hires.pp')
level_10 = iris.Constraint(model_level_number=10)
cubes = iris.load(filename, level_10)
```

Constraints can be combined using `&` to represent a more restrictive constraint to `load`:

```
filename = iris.sample_data_path('uk_hires.pp')
forecast_6 = iris.Constraint(forecast_period=6)
level_10 = iris.Constraint(model_level_number=10)
cubes = iris.load(filename, forecast_6 & level_10)
```

As well as being able to combine constraints using `&`, the *iris.Constraint* class can accept multiple arguments, and a list of values can be given to constrain a coordinate to one of a collection of values:

```
filename = iris.sample_data_path('uk_hires.pp')
level_10_or_16_fp_6 = iris.Constraint(model_level_number=[10, 16], forecast_period=6)
cubes = iris.load(filename, level_10_or_16_fp_6)
```

A common requirement is to limit the value of a coordinate to a specific range, this can be achieved by passing the constraint a function:

```
def bottom_16_levels(cell):
    # return True or False as to whether the cell in question should be kept
    return cell <= 16

filename = iris.sample_data_path('uk_hires.pp')
level_lt_16 = iris.Constraint(model_level_number=bottom_16_levels)
cubes = iris.load(filename, level_lt_16)
```

Note: As with many of the examples later in this documentation, the simple function above can be conveniently written as a lambda function on a single line:

```
bottom_16_levels = lambda cell: cell <= 16
```

Note also the *warning on equality constraints with floating point coordinates*.

Cube attributes can also be part of the constraint criteria. Supposing a cube attribute of `STASH` existed, as is the case when loading PP files, then specific STASH codes can be filtered:

```
filename = iris.sample_data_path('uk_hires.pp')
level_10_with_stash = iris.AttributeConstraint(STASH='m01s00i004') & iris.
↪Constraint(model_level_number=10)
cubes = iris.load(filename, level_10_with_stash)
```

See also:

For advanced usage there are further examples in the *iris.Constraint* reference documentation.

5.3.1 Constraining a Circular Coordinate Across its Boundary

Occasionally you may need to constrain your cube with a region that crosses the boundary of a circular coordinate (this is often the meridian or the dateline / antimeridian). An example use-case of this is to extract the entire Pacific Ocean from a cube whose longitudes are bounded by the dateline.

This functionality cannot be provided reliably using constraints. Instead you should use the functionality provided by *cube.intersection* to extract this region.

5.3.2 Constraining on Time

Iris follows NetCDF-CF rules in representing time coordinate values as normalised, purely numeric, values which are normalised by the calendar specified in the coordinate's units (e.g. "days since 1970-01-01"). However, when constraining by time we usually want to test calendar-related aspects such as hours of the day or months of the year, so Iris provides special features to facilitate this:

Firstly, when Iris evaluates Constraint expressions, it will convert time-coordinate values (points and bounds) from numbers into `datetime`-like objects for ease of calendar-based testing.

```
>>> filename = iris.sample_data_path('uk_hires.pp')
>>> cube_all = iris.load_cube(filename, 'air_potential_temperature')
>>> print('All times :\n' + str(cube_all.coord('time')))
All times :
DimCoord([2009-11-19 10:00:00, 2009-11-19 11:00:00, 2009-11-19 12:00:00], standard_name=
↪'time', calendar='gregorian')
>>> # Define a function which accepts a datetime as its argument (this is simplified in␣
↪later examples).
>>> hour_11 = iris.Constraint(time=lambda cell: cell.point.hour == 11)
>>> cube_11 = cube_all.extract(hour_11)
>>> print('Selected times :\n' + str(cube_11.coord('time')))
Selected times :
DimCoord([2009-11-19 11:00:00], standard_name='time', calendar='gregorian')
```

Secondly, the *iris.time* module provides flexible time comparison facilities. An *iris.time.PartialDateTime* object can be compared to objects such as `datetime.datetime` instances, and this comparison will then test only those 'aspects' which the PartialDateTime instance defines:

```
>>> import datetime
>>> from iris.time import PartialDateTime
>>> dt = datetime.datetime(2011, 3, 7)
>>> print(dt > PartialDateTime(year=2010, month=6))
True
>>> print(dt > PartialDateTime(month=6))
False
>>>
```

These two facilities can be combined to provide straightforward calendar-based time selections when loading or extracting data.

The previous constraint example can now be written as:

```
>>> the_11th_hour = iris.Constraint(time=iris.time.PartialDateTime(hour=11))
>>> print(iris.load_cube(
...     iris.sample_data_path('uk_hires.pp'),
...     'air_potential_temperature' & the_11th_hour).coord('time'))
DimCoord([2009-11-19 11:00:00], standard_name='time', calendar='gregorian')
```

It is common that a cube will need to be constrained between two given dates. In the following example we construct a time sequence representing the first day of every week for many years:

```
>>> print(long_ts.coord('time'))
DimCoord([2007-04-09 00:00:00, 2007-04-16 00:00:00, 2007-04-23 00:00:00,
          ...
          2010-02-01 00:00:00, 2010-02-08 00:00:00, 2010-02-15 00:00:00],
         standard_name='time', calendar='gregorian')
```

Given two dates in datetime format, we can select all points between them.

```
>>> d1 = datetime.datetime.strptime('20070715T0000Z', '%Y%m%dT%H%MZ')
>>> d2 = datetime.datetime.strptime('20070825T0000Z', '%Y%m%dT%H%MZ')
>>> st_swithuns_daterange_07 = iris.Constraint(
...     time=lambda cell: d1 <= cell.point < d2)
>>> within_st_swithuns_07 = long_ts.extract(st_swithuns_daterange_07)
>>> print(within_st_swithuns_07.coord('time'))
DimCoord([2007-07-16 00:00:00, 2007-07-23 00:00:00, 2007-07-30 00:00:00,
          2007-08-06 00:00:00, 2007-08-13 00:00:00, 2007-08-20 00:00:00],
         standard_name='time', calendar='gregorian')
```

Alternatively, we may rewrite this using *iris.time.PartialDateTime* objects.

```
>>> pdt1 = PartialDateTime(year=2007, month=7, day=15)
>>> pdt2 = PartialDateTime(year=2007, month=8, day=25)
>>> st_swithuns_daterange_07 = iris.Constraint(
...     time=lambda cell: pdt1 <= cell.point < pdt2)
>>> within_st_swithuns_07 = long_ts.extract(st_swithuns_daterange_07)
>>> print(within_st_swithuns_07.coord('time'))
DimCoord([2007-07-16 00:00:00, 2007-07-23 00:00:00, 2007-07-30 00:00:00,
          2007-08-06 00:00:00, 2007-08-13 00:00:00, 2007-08-20 00:00:00],
         standard_name='time', calendar='gregorian')
```

A more complex example might require selecting points over an annually repeating date range. We can select points within a certain part of the year, in this case between the 15th of July through to the 25th of August. By making use of

PartialDateTime this becomes simple:

```
>>> st_swithuns_daterange = iris.Constraint(
...     time=lambda cell: PartialDateTime(month=7, day=15) <= cell <␣
↪PartialDateTime(month=8, day=25))
>>> within_st_swithuns = long_ts.extract(st_swithuns_daterange)
...
>>> print(within_st_swithuns.coord('time'))
DimCoord([2007-07-16 00:00:00, 2007-07-23 00:00:00, 2007-07-30 00:00:00,
       2007-08-06 00:00:00, 2007-08-13 00:00:00, 2007-08-20 00:00:00,
       2008-07-21 00:00:00, 2008-07-28 00:00:00, 2008-08-04 00:00:00,
       2008-08-11 00:00:00, 2008-08-18 00:00:00, 2009-07-20 00:00:00,
       2009-07-27 00:00:00, 2009-08-03 00:00:00, 2009-08-10 00:00:00,
       2009-08-17 00:00:00, 2009-08-24 00:00:00], standard_name='time', calendar=
↪'gregorian')
```

Notice how the dates printed are between the range specified in the `st_swithuns_daterange` and that they span multiple years.

5.4 Strict Loading

The *iris.load_cube()* and *iris.load_cubes()* functions are similar to *iris.load()* except they can only return *one cube per constraint*. The *iris.load_cube()* function accepts a single constraint and returns a single cube. The *iris.load_cubes()* function accepts any number of constraints and returns a list of cubes (as an *iris.cube.CubeList*). Providing no constraints to *iris.load_cube()* or *iris.load_cubes()* is equivalent to requesting exactly one cube of any type.

A single cube is loaded in the following example:

```
>>> filename = iris.sample_data_path('air_temp.pp')
>>> cube = iris.load_cube(filename)
>>> print(cube)
air_temperature / (K)                (latitude: 73; longitude: 96)
     Dimension coordinates:
          latitude                           x              -
          longitude                          -              x
...
     Cell methods:
          mean                          time
```

However, when attempting to load data which would result in anything other than one cube, an exception is raised:

```
>>> filename = iris.sample_data_path('uk_hires.pp')
>>> cube = iris.load_cube(filename)
Traceback (most recent call last):
...
iris.exceptions.ConstraintMismatchError: Expected exactly one cube, found 2.
```

Note: All the load functions share many of the same features, hence multiple files could be loaded with wildcard filenames or by providing a list of filenames.

The strict nature of *iris.load_cube()* and *iris.load_cubes()* means that, when combined with constrained loading, it is possible to ensure that precisely what was asked for on load is given - otherwise an exception is raised. This fact can be utilised to make code only run successfully if the data provided has the expected criteria.

For example, suppose that code needed `air_potential_temperature` in order to run:

```
import iris
filename = iris.sample_data_path('uk_hires.pp')
air_pot_temp = iris.load_cube(filename, 'air_potential_temperature')
print(air_pot_temp)
```

Should the file not produce exactly one cube with a standard name of 'air_potential_temperature', an exception will be raised.

Similarly, supposing a routine needed both 'surface_altitude' and 'air_potential_temperature' to be able to run:

```
import iris
filename = iris.sample_data_path('uk_hires.pp')
altitude_cube, pot_temp_cube = iris.load_cubes(filename, ['surface_altitude', 'air_
↪potential_temperature'])
```

The result of *iris.load_cubes()* in this case will be a list of 2 cubes ordered by the constraints provided. Multiple assignment has been used to put these two cubes into separate variables.

Note: In Python, lists of a pre-known length and order can be exploited using *multiple assignment*:

```
>>> number_one, number_two = [1, 2]
>>> print(number_one)
1
>>> print(number_two)
2
```

CHAPTER

SIX

SAVING IRIS CUBES

Iris supports the saving of cubes and cube lists to:

- CF netCDF (version 1.7)
- GRIB edition 2 (if iris-grib is installed)
- Met Office PP

The *iris.save()* function saves one or more cubes to a file.

If the filename includes a supported suffix then Iris will use the correct saver and the keyword argument *saver* is not required.

```
>>> import iris
>>> filename = iris.sample_data_path('uk_hires.pp')
>>> cubes = iris.load(filename)
>>> iris.save(cubes, '/tmp/uk_hires.nc')
```

Warning: Saving a cube whose data has been loaded lazily (if *cube.has_lazy_data()* returns *True*) to the same file it expects to load data from will cause both the data in-memory and the data on disk to be lost.

```
cube = iris.load_cube("somefile.nc")
# The next line causes data loss in 'somefile.nc' and the cube.
iris.save(cube, "somefile.nc")
```

In general, overwriting a file which is the source for any lazily loaded data can result in corruption. Users should proceed with caution when attempting to overwrite an existing file.

6.1 Controlling the Save Process

The *iris.save()* function passes all other keywords through to the saver function defined, or automatically set from the file extension. This enables saver specific functionality to be called.

```
>>> # Save a cube to PP
>>> iris.save(cubes[0], "myfile.pp")
>>> # Save a cube list to a PP file, appending to the contents of the file
>>> # if it already exists
>>> iris.save(cubes, "myfile.pp", append=True)
>>> # Save a cube to netCDF, defaults to NETCDF4 file format
>>> iris.save(cubes[0], "myfile.nc")
```

(continues on next page)

(continued from previous page)

```
>>> # Save a cube list to netCDF, using the NETCDF3_CLASSIC storage option
>>> iris.save(cubes, "myfile.nc", netcdf_format="NETCDF3_CLASSIC")
```

See

- *iris.fileformats.netcdf.save()*
- *iris.fileformats.pp.save()*

for more details on supported arguments for the individual savers.

6.2 Customising the Save Process

When saving to GRIB or PP, the save process may be intercepted between the translation step and the file writing. This enables customisation of the output messages, based on Cube metadata if required, over and above the translations supplied by Iris.

For example, a GRIB2 message with a particular known long_name may need to be saved to a specific parameter code and type of statistical process. This can be achieved by:

```
def tweaked_messages(cube):
    for cube, grib_message in iris_grib.save_pairs_from_cube(cube):
        # post process the GRIB2 message, prior to saving
        if cube.name() == 'carefully_customised_precipitation_amount':
            gribapi.grib_set_long(grib_message, "typeOfStatisticalProcess", 1)
            gribapi.grib_set_long(grib_message, "parameterCategory", 1)
            gribapi.grib_set_long(grib_message, "parameterNumber", 1)
        yield grib_message
iris_grib.save_messages(tweaked_messages(cubes[0]), '/tmp/agrib2.grib2')
```

Similarly a PP field may need to be written out with a specific value for LBEXP. This can be achieved by:

```
def tweaked_fields(cube):
    for cube, field in iris.fileformats.pp.save_pairs_from_cube(cube):
        # post process the PP field, prior to saving
        if cube.name() == 'air_pressure':
            field.lbexp = 'meaxp'
        elif cube.name() == 'air_density':
            field.lbexp = 'meaxr'
        yield field
iris.fileformats.pp.save_fields(tweaked_fields(cubes[0]), '/tmp/app.pp')
```

6.2.1 NetCDF

NetCDF is a flexible container for metadata and cube metadata is closely related to the CF for netCDF semantics. This means that cube metadata is well represented in netCDF files, closely resembling the in memory metadata representation. Thus there is no provision for similar save customisation functionality for netCDF saving, all customisations should be applied to the cube prior to saving to netCDF.

6.3 Bespoke Saver

A bespoke saver may be written to support an alternative file format. This can be provided to the *iris.save()* function, enabling Iris to write to a different file format. Such a custom saver will need be written to meet the needs of the file format and to handle the metadata translation from cube metadata effectively.

Implementing a bespoke saver is out of scope for the user guide.

CHAPTER

SEVEN

NAVIGATING A CUBE

After loading any cube, you will want to investigate precisely what it contains. This section is all about accessing and manipulating the metadata contained within a cube.

7.1 Cube String Representations

We have already seen a basic string representation of a cube when printing:

```
>>> import iris
>>> filename = iris.sample_data_path('rotated_pole.nc')
>>> cube = iris.load_cube(filename)
>>> print(cube)
air_pressure_at_sea_level / (Pa)    (grid_latitude: 22; grid_longitude: 36)
    Dimension coordinates:
        grid_latitude                           x                   -
        grid_longitude                          -                   x
    Scalar coordinates:
        forecast_period             0.0 hours
        forecast_reference_time     2006-06-15 00:00:00
        time                        2006-06-15 00:00:00
    Attributes:
        Conventions                 CF-1.5
        STASH                       m01s16i222
        source                      Data from Met Office Unified Model 6.01
```

This representation is equivalent to passing the cube to the `str()` function. This function can be used on any Python variable to get a string representation of that variable. Similarly there exist other standard functions for interrogating your variable: `repr()`, `type()` for example:

```
print(str(cube))
print(repr(cube))
print(type(cube))
```

Other, more verbose, functions also exist which give information on **what** you can do with *any* given variable. In most cases it is reasonable to ignore anything starting with a "_" (underscore) or a "__" (double underscore):

```
dir(cube)
help(cube)
```

7.2 Working With Cubes

Every cube has a standard name, long name and units which are accessed with *Cube.standard_name*, *Cube.long_name* and *Cube.units* respectively:

```
print(cube.standard_name)
print(cube.long_name)
print(cube.units)
```

Interrogating these with the standard `type()` function will tell you that `standard_name` and `long_name` are either a string or `None`, and `units` is an instance of `iris.unit.Unit`. A more in depth discussion on the cube units and their functional effects can be found at the end of *Cube Maths*.

You can access a string representing the "name" of a cube with the *Cube.name()* method:

```
print(cube.name())
```

The result of which is **always** a string.

Each cube also has a `numpy` array which represents the phenomenon of the cube which can be accessed with the *Cube.data* attribute. As you can see the type is a `numpy n-dimensional array`:

```
print(type(cube.data))
```

Note: When loading from most file formats in Iris, the data itself is not loaded until the **first** time that the data is requested. Hence you may have noticed that running the previous command for the first time takes a little longer than it does for subsequent calls.

For this reason, when you have a large cube it is strongly recommended that you do not access the cube's data unless you need to. For convenience *shape* and *ndim* attributes exists on a cube, which can tell you the shape of the cube's data without loading it:

```
print(cube.shape)
print(cube.ndim)
```

For more on the benefits, handling and uses of lazy data, see *Real and Lazy Data*

You can change the units of a cube using the *convert_units()* method. For example:

```
cube.convert_units('celsius')
```

As well as changing the value of the *units* attribute this will also convert the values in *data*. To replace the units without modifying the data values one can change the *units* attribute directly.

Some cubes represent a processed phenomenon which are represented with cell methods, these can be accessed on a cube with the *Cube.cell_methods* attribute:

```
print(cube.cell_methods)
```

7.3 Accessing Coordinates on the Cube

A cube's coordinates can be retrieved via *Cube.coords*. A simple for loop over the coords can print a coordinate's *name()*:

```
for coord in cube.coords():
    print(coord.name())
```

Alternatively, we can use *list comprehension* to store the names in a list:

```
coord_names = [coord.name() for coord in cube.coords()]
```

The result is a basic Python list which could be sorted alphabetically and joined together:

```
>>> print(', '.join(sorted(coord_names)))
forecast_period, forecast_reference_time, grid_latitude, grid_longitude, time
```

To get an individual coordinate given its name, the *Cube.coord* method can be used:

```
coord = cube.coord('grid_latitude')
print(type(coord))
```

Every coordinate has a *Coord.standard_name*, *Coord.long_name*, and *Coord.units* attribute:

```
print(coord.standard_name)
print(coord.long_name)
print(coord.units)
```

Additionally every coordinate can provide its *points* and *bounds* numpy array. If the coordinate has no bounds `None` will be returned:

```
print(type(coord.points))
print(type(coord.bounds))
```

7.4 Adding Metadata to a Cube

We can add and remove coordinates via *Cube.add_dim_coord*, *Cube.add_aux_coord*, and *Cube.remove_coord*.

```
>>> import iris.coords
>>> new_coord = iris.coords.AuxCoord(1, long_name='my_custom_coordinate', units='no_unit
↪')
>>> cube.add_aux_coord(new_coord)
>>> print(cube)
air_pressure_at_sea_level / (Pa)    (grid_latitude: 22; grid_longitude: 36)
    Dimension coordinates:
        grid_latitude                           x                   -
        grid_longitude                          -                   x
    Scalar coordinates:
        forecast_period             0.0 hours
        forecast_reference_time     2006-06-15 00:00:00
        my_custom_coordinate        1
        time                        2006-06-15 00:00:00
```

(continues on next page)

(continued from previous page)

```
    Attributes:
        Conventions                 CF-1.5
        STASH                       m01s16i222
        source                      Data from Met Office Unified Model 6.01
```

The coordinate `my_custom_coordinate` now exists on the cube and is listed under the non-dimensioned single valued scalar coordinates.

7.5 Adding and Removing Metadata to the Cube at Load Time

Sometimes when loading a cube problems occur when the amount of metadata is more or less than expected. This is often caused by one of the following:

- The file does not contain enough metadata, and therefore the cube cannot know everything about the file.
- Some of the metadata of the file is contained in the filename, but is not part of the actual file.
- There is not enough metadata loaded from the original file as Iris has not handled the format fully. *(in which case, please let us know about it)*

To solve this, all of *iris.load()*, *iris.load_cube()*, and *iris.load_cubes()* support a callback keyword.

The callback is a user defined function which must have the calling sequence `function(cube, field, filename)` which can make any modifications to the cube in-place, or alternatively return a completely new cube instance.

Suppose we wish to load a lagged ensemble dataset from the Met Office's GloSea4 model. The data for this example represents 13 ensemble members of 6 one month timesteps; the logistics of the model mean that the run is spread over several days.

If we try to load the data directly for `surface_temperature`:

```
>>> filename = iris.sample_data_path('GloSea4', '*.pp')
>>> print(iris.load(filename, 'surface_temperature'))
0: surface_temperature / (K)           (time: 6; forecast_reference_time: 2; latitude:␣
↪145; longitude: 192)
1: surface_temperature / (K)           (time: 6; forecast_reference_time: 2; latitude:␣
↪145; longitude: 192)
2: surface_temperature / (K)           (realization: 9; time: 6; latitude: 145;␣
↪longitude: 192)
```

We get multiple cubes some with more dimensions than expected, some without a `realization` (i.e. ensemble member) dimension. In this case, two of the PP files have been encoded without the appropriate `realization` number attribute, which means that the appropriate coordinate cannot be added to the resultant cube. Fortunately, the missing attribute has been encoded in the filename which, given the filename, we could extract:

```
filename = iris.sample_data_path('GloSea4', 'ensemble_001.pp')
realization = int(filename[-6:-3])
print(realization)
```

We can solve this problem by adding the appropriate metadata, on load, by using a callback function, which runs on a field by field basis *before* they are automatically merged together:

```
import numpy as np
import iris
```

(continues on next page)

(continued from previous page)

```
import iris.coords as icoords

def lagged_ensemble_callback(cube, field, filename):
    # Add our own realization coordinate if it doesn't already exist.
    if not cube.coords('realization'):
        realization = np.int32(filename[-6:-3])
        ensemble_coord = icoords.AuxCoord(realization, standard_name='realization',␣
↪units="1")
        cube.add_aux_coord(ensemble_coord)

filename = iris.sample_data_path('GloSea4', '*.pp')

print(iris.load(filename, 'surface_temperature', callback=lagged_ensemble_callback))
```

The result is a single cube which represents the data in a form that was expected:

```
0: surface_temperature / (K)           (realization: 13; time: 6; latitude: 145;␣
↪longitude: 192)
```

CHAPTER

EIGHT

SUBSETTING A CUBE

The *Loading Iris Cubes* section of the user guide showed how to load data into multidimensional Iris cubes. However it is often necessary to reduce the dimensionality of a cube down to something more appropriate and/or manageable.

Iris provides several ways of reducing both the amount of data and/or the number of dimensions in your cube depending on the circumstance. In all cases **the subset of a valid cube is itself a valid cube**.

8.1 Cube Extraction

A subset of a cube can be "extracted" from a multi-dimensional cube in order to reduce its dimensionality:

```
>>> import iris
>>> filename = iris.sample_data_path('space_weather.nc')
>>> cube = iris.load_cube(filename, 'electron density')
>>> equator_slice = cube.extract(iris.Constraint(grid_latitude=0))
>>> print(equator_slice)
electron density / (1E11 e/m^3)     (height: 29; grid_longitude: 31)
    Dimension coordinates:
        height                           x                  -
        grid_longitude                   -                  x
    Auxiliary coordinates:
        latitude                         -                  x
        longitude                        -                  x
    Scalar coordinates:
        grid_latitude               0.0 degrees
    Attributes:
        Conventions                 CF-1.5
```

In this example we start with a 3 dimensional cube, with dimensions of `height`, `grid_latitude` and `grid_longitude`, and extract every point where the latitude is 0, resulting in a 2d cube with axes of `height` and `grid_longitude`.

Warning: Caution is required when using equality constraints with floating point coordinates such as `grid_latitude`. Printing the points of a coordinate does not necessarily show the full precision of the underlying number and it is very easy return no matches to a constraint when one was expected. This can be avoided by using a function as the argument to the constraint:

```
def near_zero(cell):
    """Returns true if the cell is between -0.1 and 0.1."""
    return -0.1 < cell < 0.1

equator_constraint = iris.Constraint(grid_latitude=near_zero)
```

Often you will see this construct in shorthand using a lambda function definition:

```
equator_constraint = iris.Constraint(grid_latitude=lambda cell: -0.1 < cell < 0.1)
```

The extract method could be applied again to the *equator_slice* cube to get a further subset.

For example to get a `height` of 9000 metres at the equator the following line extends the previous example:

```
equator_height_9km_slice = equator_slice.extract(iris.Constraint(height=9000))
print(equator_height_9km_slice)
```

The two steps required to get `height` of 9000 m at the equator can be simplified into a single constraint:

```
equator_height_9km_slice = cube.extract(iris.Constraint(grid_latitude=0, height=9000))
print(equator_height_9km_slice)
```

As we saw in *Loading Iris Cubes* the result of *iris.load()* is a *CubeList*. The `extract` method also exists on a *CubeList* and behaves in exactly the same way as loading with constraints:

```
>>> import iris
>>> air_temp_and_fp_6 = iris.Constraint('air_potential_temperature', forecast_period=6)
>>> level_10 = iris.Constraint(model_level_number=10)
>>> filename = iris.sample_data_path('uk_hires.pp')
>>> cubes = iris.load(filename).extract(air_temp_and_fp_6 & level_10)
>>> print(cubes)
0: air_potential_temperature / (K)     (grid_latitude: 204; grid_longitude: 187)
>>> print(cubes[0])
air_potential_temperature / (K)     (grid_latitude: 204; grid_longitude: 187)
    Dimension coordinates:
        grid_latitude                           x                   -
        grid_longitude                          -                   x
    Auxiliary coordinates:
        surface_altitude                        x                   x
    Derived coordinates:
        altitude                                x                   x
    Scalar coordinates:
        forecast_period             6.0 hours
        forecast_reference_time     2009-11-19 04:00:00
        level_height                395.0 m, bound=(360.0, 433.3332) m
        model_level_number          10
        sigma                       0.9549927, bound=(0.9589389, 0.95068014)
        time                        2009-11-19 10:00:00
    Attributes:
        STASH                       m01s00i004
        source                      Data from Met Office Unified Model
        um_version                  7.3
```

8.2 Cube Iteration

It is not possible to directly iterate over an Iris cube. That is, you cannot use code such as `for x in cube:`. However, you can iterate over cube slices, as this section details.

A useful way of dealing with a Cube in its **entirety** is by iterating over its layers or slices. For example, to deal with a 3 dimensional cube (z,y,x) you could iterate over all 2 dimensional slices in y and x which make up the full 3d cube.:

```
import iris
filename = iris.sample_data_path('hybrid_height.nc')
cube = iris.load_cube(filename)
print(cube)
for yx_slice in cube.slices(['grid_latitude', 'grid_longitude']):
   print(repr(yx_slice))
```

As the original cube had the shape (15, 100, 100) there were 15 latitude longitude slices and hence the line `print(repr(yx_slice))` was run 15 times.

Note: The order of latitude and longitude in the list is important; had they been swapped the resultant cube slices would have been transposed.

For further information see *Cube.slices*.

This method can handle n-dimensional slices by providing more or fewer coordinate names in the list to **slices**:

```
import iris
filename = iris.sample_data_path('hybrid_height.nc')
cube = iris.load_cube(filename)
print(cube)
for i, x_slice in enumerate(cube.slices(['grid_longitude'])):
   print(i, repr(x_slice))
```

The Python function `enumerate()` is used in this example to provide an incrementing variable **i** which is printed with the summary of each cube slice. Note that there were 1500 1d longitude cubes as a result of slicing the 3 dimensional cube (15, 100, 100) by longitude (i starts at 0 and 1500 = 15 * 100).

Hint: It is often useful to get a single 2d slice from a multidimensional cube in order to develop a 2d plot function, for example. This can be achieved by using the `next()` function on the result of slices:

```
first_slice = next(cube.slices(['grid_latitude', 'grid_longitude']))
```

Once the your code can handle a 2d slice, it is then an easy step to loop over **all** 2d slices within the bigger cube using the slices method.

8.3 Cube Indexing

In the same way that you would expect a numeric multidimensional array to be **indexed** to take a subset of your original array, you can **index** a Cube for the same purpose.

Here are some examples of array indexing in numpy:

```
import numpy as np
# create an array of 12 consecutive integers starting from 0
a = np.arange(12)
print(a)

print(a[0])     # first element of the array

print(a[-1])    # last element of the array

print(a[0:4])   # first four elements of the array (the same as a[:4])

print(a[-4:])   # last four elements of the array

print(a[::-1])  # gives all of the array, but backwards

# Make a 2d array by reshaping a
b = a.reshape(3, 4)
print(b)

print(b[0, 0])  # first element of the first and second dimensions

print(b[0])     # first element of the first dimension (+ every other dimension)

# get the second element of the first dimension and all of the second dimension
# in reverse, by steps of two.
print(b[1, ::-2])
```

Similarly, Iris cubes have indexing capability:

```
import iris
filename = iris.sample_data_path('hybrid_height.nc')
cube = iris.load_cube(filename)

print(cube)

# get the first element of the first dimension (+ every other dimension)
print(cube[0])

# get the last element of the first dimension (+ every other dimension)
print(cube[-1])

# get the first 4 elements of the first dimension (+ every other dimension)
print(cube[0:4])

# Get the first element of the first and third dimension (+ every other dimension)
print(cube[0, :, 0])
```

(continues on next page)

(continued from previous page)

```
# Get the second element of the first dimension and all of the second dimension
# in reverse, by steps of two.
print(cube[1, ::-2])
```

CHAPTER

NINE

REAL AND LAZY DATA

We have seen in the *Iris Data Structures* section of the user guide that Iris cubes contain data and metadata about a phenomenon. The data element of a cube is always an array, but the array may be either "real" or "lazy".

In this section of the user guide we will look specifically at the concepts of real and lazy data as they apply to the cube and other data structures in Iris.

9.1 What is Real and Lazy Data?

In Iris, we use the term **real data** to describe data arrays that are loaded into memory. Real data is typically provided as a NumPy array, which has a shape and data type that are used to describe the array's data points. Each data point takes up a small amount of memory, which means large NumPy arrays can take up a large amount of memory.

Conversely, we use the term **lazy data** to describe data that is not loaded into memory. (This is sometimes also referred to as **deferred data**.) In Iris, lazy data is provided as a dask array. A dask array also has a shape and data type but the dask array's data points remain on disk and only loaded into memory in small chunks when absolutely necessary. This has key performance benefits for handling large amounts of data, where both calculation time and storage requirements can be significantly reduced.

In Iris, when actual data values are needed from a lazy data array, it is *'realised'* : this means that all the actual values are read in from the file, and a 'real' (i.e. numpy) array replaces the lazy array within the Iris object.

Following realisation, the Iris object just contains the actual ('real') data, so the time cost of reading all the data is not incurred again. From here on, access to the data is fast, but it now occupies its full memory space.

In particular, any direct reference to a *cube.data* will realise the cube data content : any lazy content is lost as the data is read from file, and the cube content is replaced with a real array. This is also referred to simply as "touching" the data.

See the section *When Does My Data Become Real?* for more examples of this.

You can check whether a cube has real data or lazy data by using the method *has_lazy_data()*. For example:

```
>>> cube = iris.load_cube(iris.sample_data_path('air_temp.pp'))
>>> cube.has_lazy_data()
True
# Realise the lazy data.
>>> cube.data
>>> cube.has_lazy_data()
False
```

9.2 Benefits

The primary advantage of using lazy data is that it enables out-of-core processing; that is, the loading and manipulating of datasets without loading the full data into memory.

There are two key benefits from this :

Firstly, the result of a calculation on a large dataset often occupies much less storage space than the source data – such as for instance a maximum data value calculated over a large number of datafiles. In these cases the result can be computed in sections, without ever requiring the entire source dataset to be loaded, thus drastically reducing memory footprint. This strategy of task division can also enable reduced execution time through the effective use of parallel processing capabilities.

Secondly, it is often simply convenient to form a calculation on a large dataset, of which only a certain portion is required at any one time – for example, plotting individual timesteps from a large sequence. In such cases, a required portion can be extracted and realised without calculating the entire result.

9.3 When Does My Data Become Real?

Certain operations, such as cube indexing and statistics, can be performed in a lazy fashion, producing a 'lazy' result from a lazy input, so that no realisation immediately occurs. However other operations, such as plotting or printing data values, will always trigger the 'realisation' of data.

When you load a dataset using Iris the data array will almost always initially be a lazy array. This section details some operations that will realise lazy data as well as some operations that will maintain lazy data. We use the term **realise** to mean converting lazy data into real data.

Most operations on data arrays can be run equivalently on both real and lazy data. If the data array is real then the operation will be run on the data array immediately. The results of the operation will be available as soon as processing is completed. If the data array is lazy then the operation will be deferred and the data array will remain lazy until you request the result (such as when you read from `cube.data`):

```
>>> cube = iris.load_cube(iris.sample_data_path('air_temp.pp'))
>>> cube.has_lazy_data()
True
>>> cube += 5
>>> cube.has_lazy_data()
True
```

The process by which the operation is deferred until the result is requested is referred to as **lazy evaluation**.

Certain operations, including regridding and plotting, can only be run on real data. Calling such operations on lazy data will automatically realise your lazy data.

You can also realise (and so load into memory) your cube's lazy data if you 'touch' the data. To 'touch' the data means directly accessing the data by calling `cube.data`, as in the previous example.

9.3.1 Core Data

Cubes have the concept of "core data". This returns the cube's data in its current state:

- If a cube has lazy data, calling the cube's *core_data()* method will return the cube's lazy dask array. Calling the cube's *core_data()* method **will never realise** the cube's data.
- If a cube has real data, calling the cube's *core_data()* method will return the cube's real NumPy array.

For example:

```
>>> cube = iris.load_cube(iris.sample_data_path('air_temp.pp'))
>>> cube.has_lazy_data()
True

>>> the_data = cube.core_data()
>>> type(the_data)
<class 'dask.array.core.Array'>
>>> cube.has_lazy_data()
True

# Realise the lazy data.
>>> cube.data
>>> the_data = cube.core_data()
>>> type(the_data)
<type 'numpy.ndarray'>
>>> cube.has_lazy_data()
False
```

9.4 Coordinates

In the same way that Iris cubes contain a data array, Iris coordinates contain a points array and an optional bounds array. Coordinate points and bounds arrays can also be real or lazy:

- A *DimCoord* will only ever have **real** points and bounds arrays because of monotonicity checks that realise lazy arrays.
- An *AuxCoord* can have **real or lazy** points and bounds.
- An *AuxCoordFactory* (or derived coordinate) can have **real or lazy** points and bounds. If all of the *AuxCoord* instances used to construct the derived coordinate have real points and bounds then the derived coordinate will have real points and bounds, otherwise the derived coordinate will have lazy points and bounds.

Iris cubes and coordinates have very similar interfaces, which extends to accessing coordinates' lazy points and bounds:

```
>>> cube = iris.load_cube(iris.sample_data_path('hybrid_height.nc'), 'air_potential_
↪temperature')

>>> dim_coord = cube.coord('model_level_number')
>>> print(dim_coord.has_lazy_points())
False
>>> print(dim_coord.has_bounds())
False
>>> print(dim_coord.has_lazy_bounds())
False
```

(continues on next page)

(continued from previous page)

```
>>> aux_coord = cube.coord('sigma')
>>> print(aux_coord.has_lazy_points())
True
>>> print(aux_coord.has_bounds())
True
>>> print(aux_coord.has_lazy_bounds())
True

# Realise the lazy points. This will **not** realise the lazy bounds.
>>> points = aux_coord.points
>>> print(aux_coord.has_lazy_points())
False
>>> print(aux_coord.has_lazy_bounds())
True

>>> derived_coord = cube.coord('altitude')
>>> print(derived_coord.has_lazy_points())
True
>>> print(derived_coord.has_bounds())
True
>>> print(derived_coord.has_lazy_bounds())
True
```

Note: Printing a lazy *AuxCoord* will realise its points and bounds arrays!

9.5 Dask Processing Options

Iris uses dask to provide lazy data arrays for both Iris cubes and coordinates, and for computing deferred operations on lazy arrays.

Dask provides processing options to control how deferred operations on lazy arrays are computed. This is provided via the `dask.set_options` interface. See the dask documentation for more information on setting dask processing options.

CHAPTER

TEN

PLOTTING A CUBE

Iris utilises the power of Python's Matplotlib package in order to generate high quality, production ready 1D and 2D plots. The functionality of the Matplotlib pyplot module has been extended within Iris to facilitate easy visualisation of a cube's data.

10.1 Matplotlib's Pyplot Basics

A simple line plot can be created using the `matplotlib.pyplot.plot()` function:

```
import matplotlib.pyplot as plt
plt.plot([1, 2, 2.5])
plt.show()
```

This code will automatically create a figure with appropriate axes for the plot and show it on screen. The call to **plt.plot([1, 2, 2.5])** will create a line plot with appropriate axes for the data (x=0, y=1; x=1, y=2; x=2, y=2.5). The call to **plt.show()** tells Matplotlib that you have finished with this plot and that you would like to visualise it in a window. This is an example of using matplotlib in *non-interactive* mode.

There are two modes of rendering within Matplotlib; **interactive** and **non-interactive**.

10.1.1 Interactive Plot Rendering

The previous example was *non-interactive* as the figure is only rendered *after* the call to `plt.show()`. Rendering plots *interactively* can be achieved by changing the interactive mode:

```
import matplotlib.pyplot as plt
plt.interactive(True)
plt.plot([1, 2, 2.5])
```

In this case the plot is rendered automatically with no need to explicitly call `matplotlib.pyplot.show()` after **plt.plot**. Subsequent changes to your figure will be automatically rendered in the window.

The current rendering mode can be determined as follows:

```
import matplotlib.pyplot as plt
print(plt.isinteractive())
```

Note: For clarity, each example includes all of the imports required to run on its own; when combining examples such as the two above, it would not be necessary to repeat the import statement more than once:

```
import matplotlib.pyplot as plt
plt.interactive(True)
plt.plot([1, 2, 2.5])
print(plt.isinteractive())
```

Interactive mode does not clear out the figure buffer, so figures have to be explicitly closed when they are finished with:

```
plt.close()
```

– or just close the figure window.

Interactive mode sometimes requires an extra draw command to update all changes, which can be done with:

```
plt.draw()
```

For the remainder of this tutorial we will work in non-interactive mode, so ensure that interactive mode is turned off with:

```
plt.interactive(False)
```

10.1.2 Saving a Plot

The `matplotlib.pyplot.savefig()` function is similar to **plt.show()** in that they are both *non-interactive* visualisation modes. As you might expect, **plt.savefig** saves your figure as an image:

```
import matplotlib.pyplot as plt
plt.plot([1, 2, 2.5])
plt.savefig('plot123.png')
```

The filename extension passed to the `matplotlib.pyplot.savefig()` function can be used to control the output file format of the plot (keywords can also be used to control this and other aspects, see `matplotlib.pyplot.savefig()`).

Some of the formats which are supported by **plt.savefig**:

Format	Type	Description
EPS	Vector	Encapsulated PostScript
PDF	Vector	Portable Document Format
PNG	Raster	Portable Network Graphics, a format with a lossless compression method
PS	Vector	PostScript, ideal for printer output
SVG	Vector	Scalable Vector Graphics, XML based

10.2 Iris Cube Plotting

The Iris modules *`iris.quickplot`* and *`iris.plot`* extend the Matplotlib pyplot interface by implementing thin *wrapper* functions. These wrapper functions simply bridge the gap between an Iris cube and the data expected by standard Matplotlib pyplot functions. This means that *all* Matplotlib pyplot functionality, including keyword options, are still available through the Iris plotting wrapper functions.

As a rule of thumb:

- if you wish to do a visualisation with a cube, use `iris.plot` or `iris.quickplot`.

- if you wish to show, save or manipulate **any** visualisation, including ones created with Iris, use `matplotlib.pyplot`.
- if you wish to create a non cube visualisation, also use `matplotlib.pyplot`.

The `iris.quickplot` module is exactly the same as the `iris.plot` module, except that `quickplot` will add a title, x and y labels and a colorbar where appropriate.

Note: In all subsequent examples the `matplotlib.pyplot`, `iris.plot` and `iris.quickplot` modules are imported as `plt`, `iplt` and `qplt` respectively in order to make the code more readable. This is equivalent to:

```
import matplotlib.pyplot as plt
import iris.plot as iplt
import iris.quickplot as qplt
```

10.2.1 Plotting 1-Dimensional Cubes

The simplest 1D plot is achieved with the *iris.plot.plot()* function. The syntax is very similar to that which you would provide to Matplotlib's equivalent `matplotlib.pyplot.plot()` and indeed all of the keyword arguments are equivalent:

```
import matplotlib.pyplot as plt

import iris
import iris.plot as iplt

fname = iris.sample_data_path("air_temp.pp")
temperature = iris.load_cube(fname)

# Take a 1d slice using array style indexing.
temperature_1d = temperature[5, :]

iplt.plot(temperature_1d)

plt.show()
```

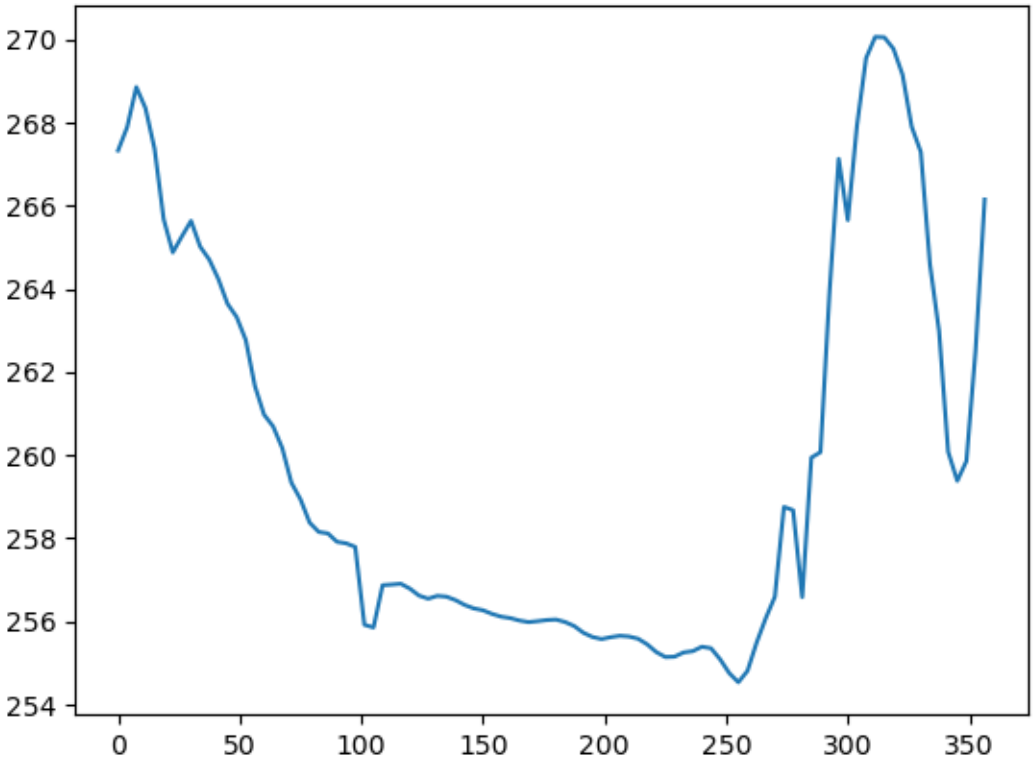
For more information on how this example reduced the 2D cube to 1 dimension see the previous section entitled *Subsetting a Cube*.

Note: Axis labels and a plot title can be added using the `plt.title()`, `plt.xlabel()` and `plt.ylabel()` functions.

As well as providing simple Matplotlib wrappers, Iris also has a *iris.quickplot* module, which adds extra cube based metadata to a plot. For example, the previous plot can be improved quickly by replacing **iris.plot** with **iris.quickplot**:

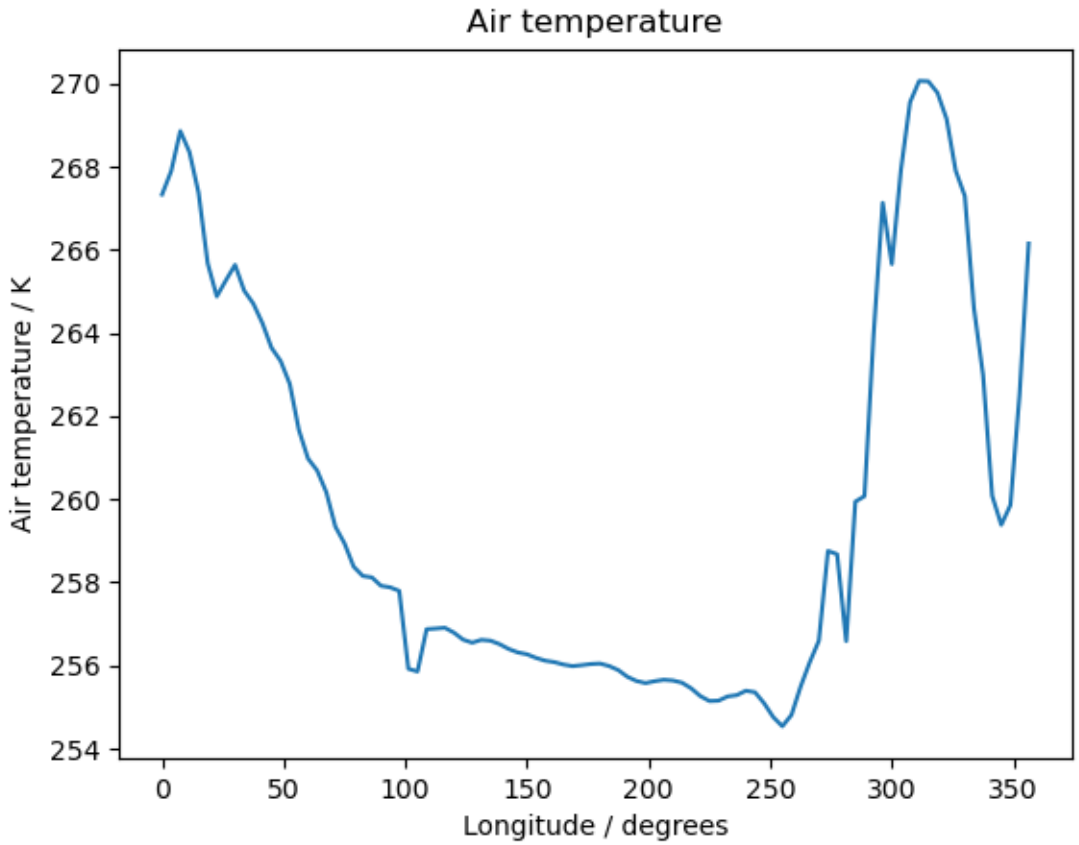
```
import matplotlib.pyplot as plt

import iris
import iris.quickplot as qplt

fname = iris.sample_data_path("air_temp.pp")
temperature = iris.load_cube(fname)
```

(continues on next page)


270
268
266
264
262
260
258
256
254
0
50
100
150
200
250
300
350

(continued from previous page)

```
# Take a 1d slice using array style indexing.
temperature_1d = temperature[5, :]

qplt.plot(temperature_1d)

plt.show()
```



Multi-Line Plot

A multi-lined (or over-plotted) plot, with a legend, can be achieved easily by calling *iris.plot.plot()* or *iris.quickplot.plot()* consecutively and providing the label keyword to identify it. Once all of the lines have been added the `matplotlib.pyplot.legend()` function can be called to indicate that a legend is desired:

```
"""
Multi-Line Temperature Profile Plot
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

"""

import matplotlib.pyplot as plt
```

(continues on next page)

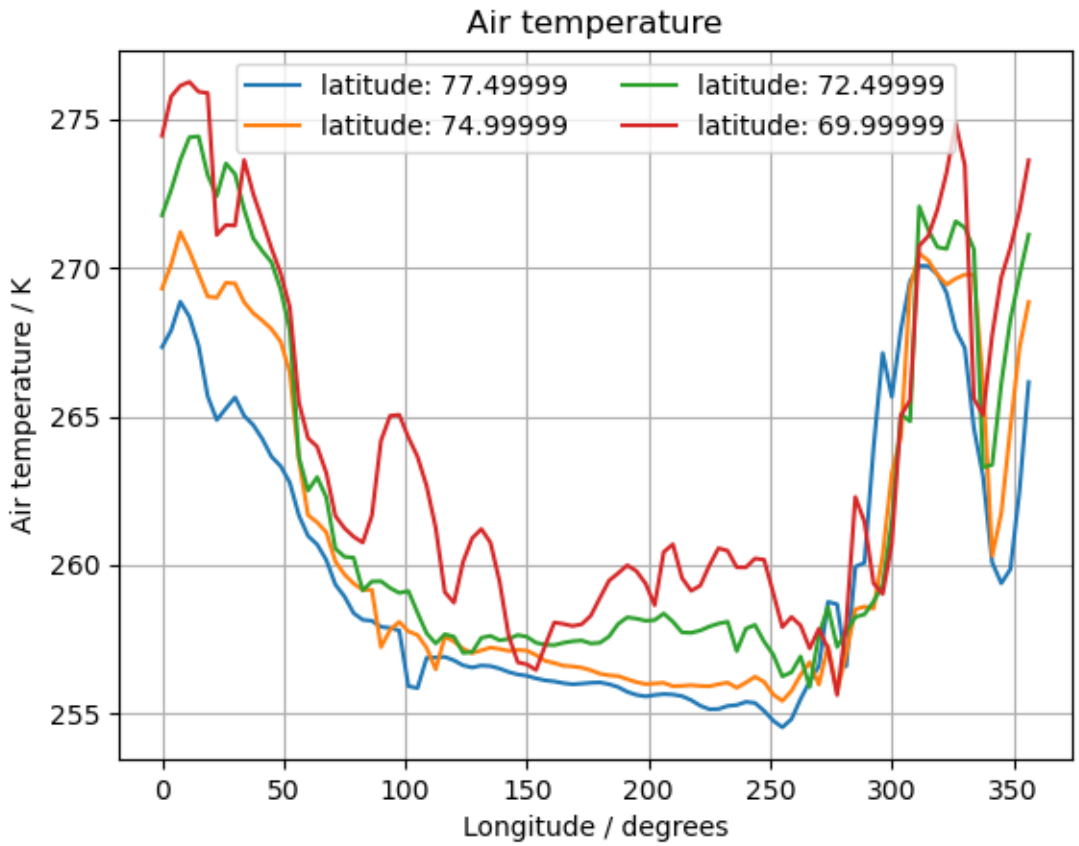
(continued from previous page)

```
import iris
import iris.plot as iplt
import iris.quickplot as qplt


def main():
    fname = iris.sample_data_path("air_temp.pp")

    # Load exactly one cube from the given file.
    temperature = iris.load_cube(fname)

    # We only want a small number of latitudes, so filter some out
    # using "extract".
    temperature = temperature.extract(
        iris.Constraint(latitude=lambda cell: 68 <= cell < 78)
    )

    for cube in temperature.slices("longitude"):

        # Create a string label to identify this cube (i.e. latitude: value).
        cube_label = "latitude: %s" % cube.coord("latitude").points[0]

        # Plot the cube, and associate it with a label.
        qplt.plot(cube, label=cube_label)

    # Add the legend with 2 columns.
    plt.legend(ncol=2)

    # Put a grid on the plot.
    plt.grid(True)

    # Tell matplotlib not to extend the plot axes range to nicely
    # rounded numbers.
    plt.axis("tight")

    # Finally, show it.
    iplt.show()


if __name__ == "__main__":
    main()
```

This example of consecutive `qplt.plot` calls coupled with the *`Cube.slices()`* method on a cube shows the temperature at some latitude cross-sections.

Note: The previous example uses the `if __name__ == "__main__"` style to run the desired code if and only if the script is run from the command line.

This is a good habit to get into when writing scripts in Python as it means that any useful functions or variables defined within the script can be imported into other scripts without running all of the code and thus creating an unwanted plot. This is discussed in more detail at http://effbot.org/pyfaq/tutor-what-is-if-name-main-for.htm.


Air temperature
latitude: 77.49999
latitude: 74.99999
latitude: 72.49999
latitude: 69.99999
Air temperature / K
275
270
265
260
255
0
50
100
150
200
250
300
350
Longitude / degrees

In order to run this example, you will need to copy the code into a file and run it using `python my_file.py`.

10.2.2 Plotting 2-Dimensional Cubes

Creating Maps

Whenever a 2D plot is created using an *iris.coord_systems.CoordSystem*, a cartopy `GeoAxes` instance is created, which can be accessed with the `matplotlib.pyplot.gca()` function.

Given the current map, you can draw gridlines and coastlines amongst other things.

See also:

`cartopy's gridlines()`, `cartopy's coastlines()`.

Cube Contour

A simple contour plot of a cube can be created with either the *iris.plot.contour()* or *iris.quickplot.contour()* functions:

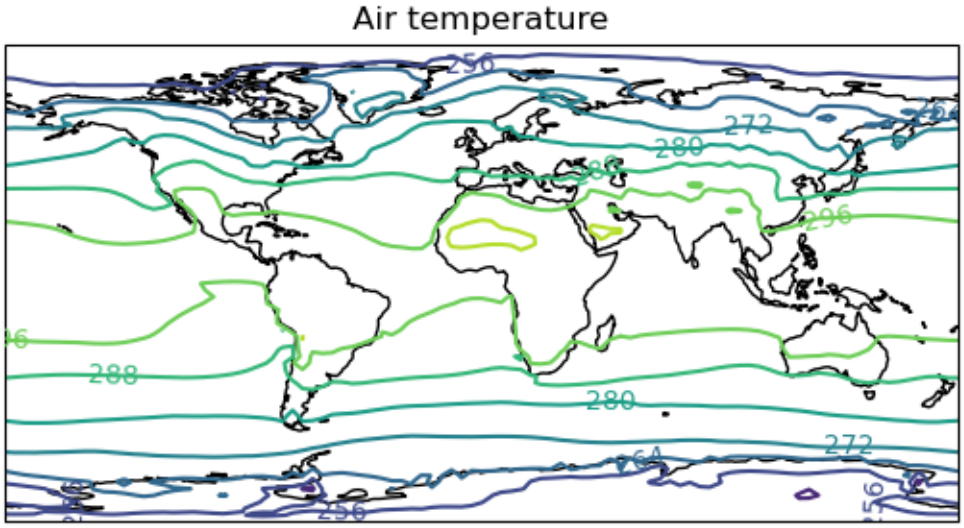
```
import matplotlib.pyplot as plt

import iris
import iris.quickplot as qplt

fname = iris.sample_data_path("air_temp.pp")
temperature_cube = iris.load_cube(fname)

# Add a contour, and put the result in a variable called contour.
contour = qplt.contour(temperature_cube)

# Add coastlines to the map created by contour.
plt.gca().coastlines()

# Add contour labels based on the contour we have just created.
plt.clabel(contour, inline=False)

plt.show()
```

Cube Filled Contour

Similarly a filled contour plot of a cube can be created with the *iris.plot.contourf()* or *iris.quickplot.contourf()* functions:

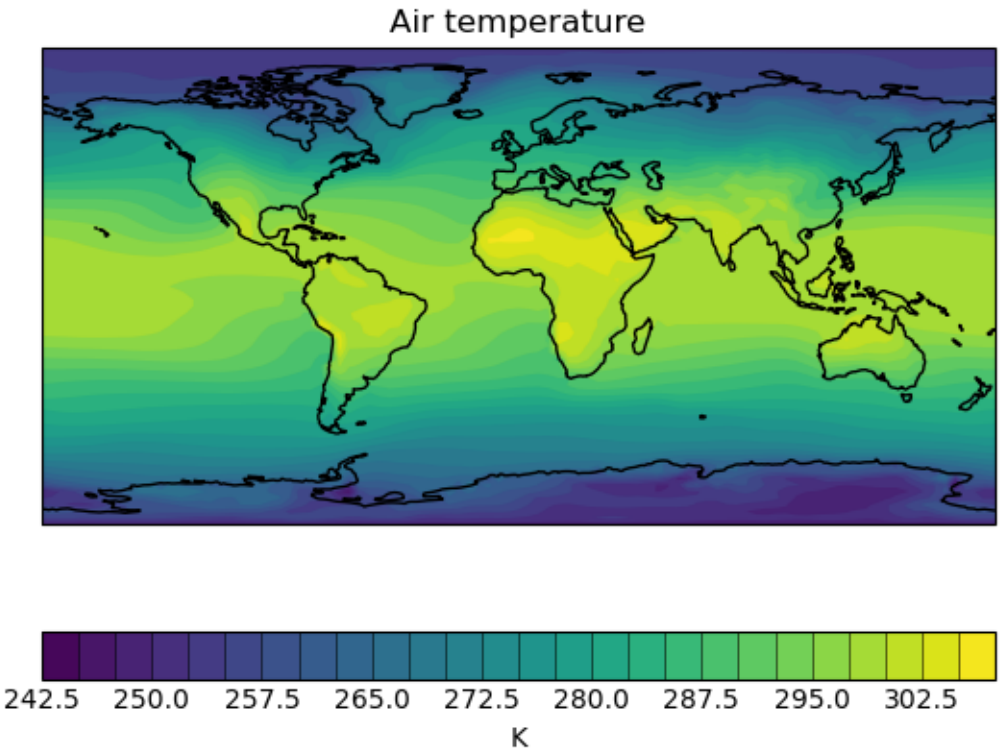
```
import matplotlib.pyplot as plt

import iris
import iris.quickplot as qplt

fname = iris.sample_data_path("air_temp.pp")
temperature_cube = iris.load_cube(fname)
```

(continues on next page)


Air temperature
256
272
280
296
288
280
272
256

(continued from previous page)

```
# Draw the contour with 25 levels.
qplt.contourf(temperature_cube, 25)

# Add coastlines to the map created by contourf.
plt.gca().coastlines()

plt.show()
```

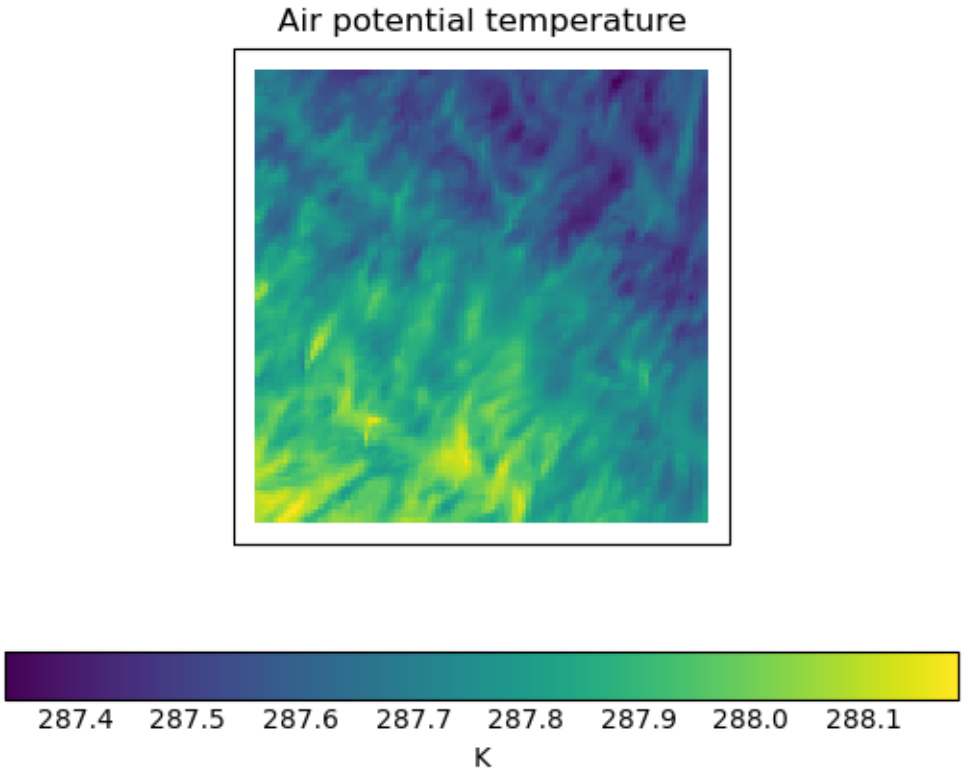


Cube Block Plot

In some situations the underlying coordinates are better represented with a continuous bounded coordinate, in which case a "block" plot may be more appropriate. Continuous block plots can be achieved with either *iris.plot.pcolormesh()* or *iris.quickplot.pcolormesh()*.

Note: If the cube's coordinates do not have bounds, *iris.plot.pcolormesh()* and *iris.quickplot.pcolormesh()* will attempt to guess suitable values based on their points (see also *iris.coords.Coord.guess_bounds()*).

```
import matplotlib.pyplot as plt

import iris
import iris.quickplot as qplt

# Load the data for a single value of model level number.
fname = iris.sample_data_path("hybrid_height.nc")
temperature_cube = iris.load_cube(fname, iris.Constraint(model_level_number=1))

# Draw the block plot.
qplt.pcolormesh(temperature_cube)

plt.show()
```



10.3 Brewer Colour Palettes

Iris includes colour specifications and designs developed by Cynthia Brewer These colour schemes are freely available under the following licence:

```
Apache-Style Software License for ColorBrewer software and ColorBrewer Color Schemes

Copyright (c) 2002 Cynthia Brewer, Mark Harrower, and The Pennsylvania State University.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this␣
↪file except in compliance with the License.
You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed
under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
CONDITIONS OF ANY KIND, either express or implied. See the License for the
specific language governing permissions and limitations under the License.
```

To include a reference in a journal article or report please refer to section 5 in the citation guidance provided by Cynthia Brewer.

For adding citations to Iris plots, see *Adding a Citation* (below).

10.3.1 Available Brewer Schemes

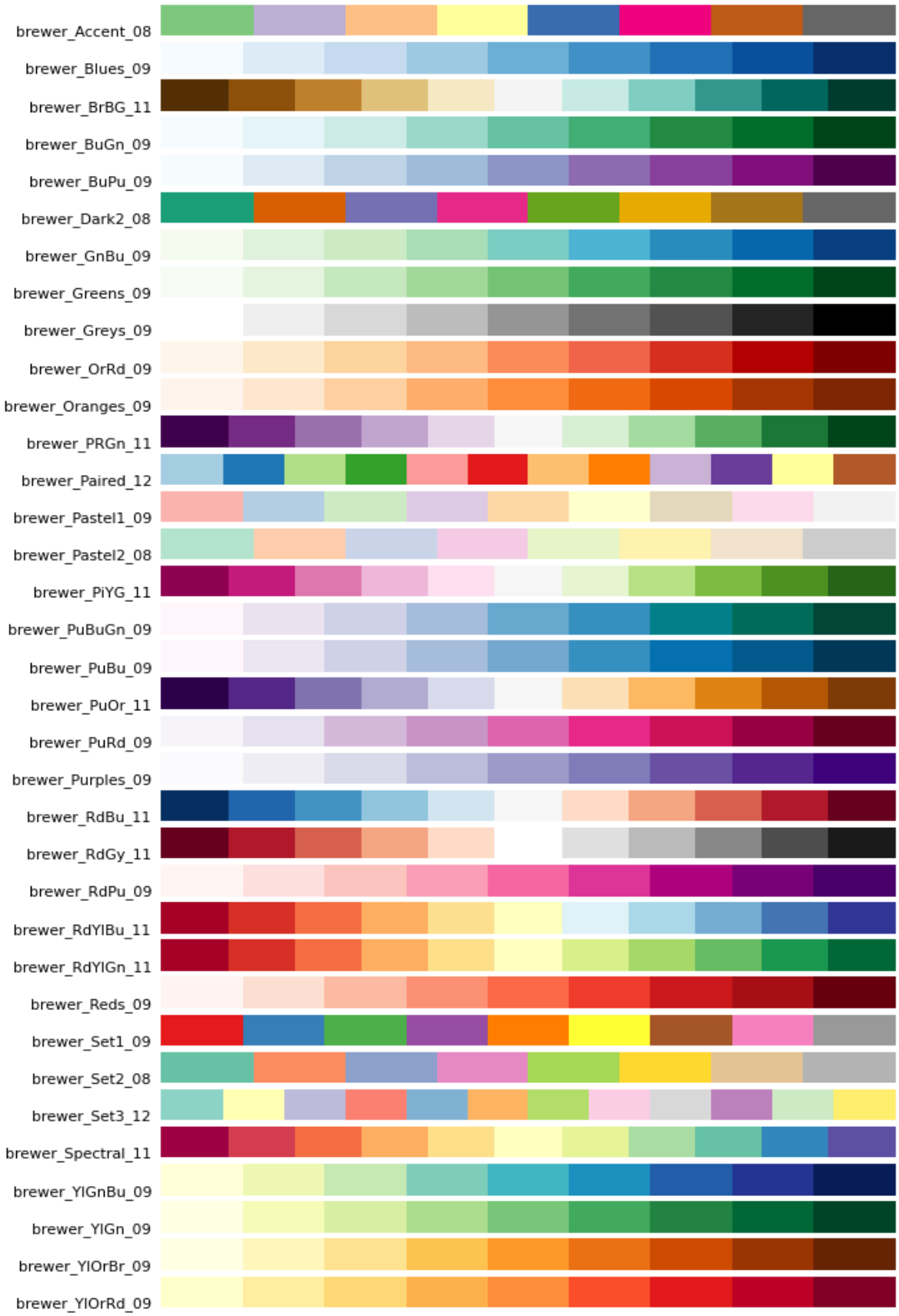
The following subset of Brewer palettes found at colorbrewer2.org are available within Iris.

10.3.2 Plotting With Brewer

To plot a cube using a Brewer colour palette, simply select one of the Iris registered Brewer colour palettes and plot the cube as normal. The Brewer palettes become available once *iris.plot* or *iris.quickplot* are imported.

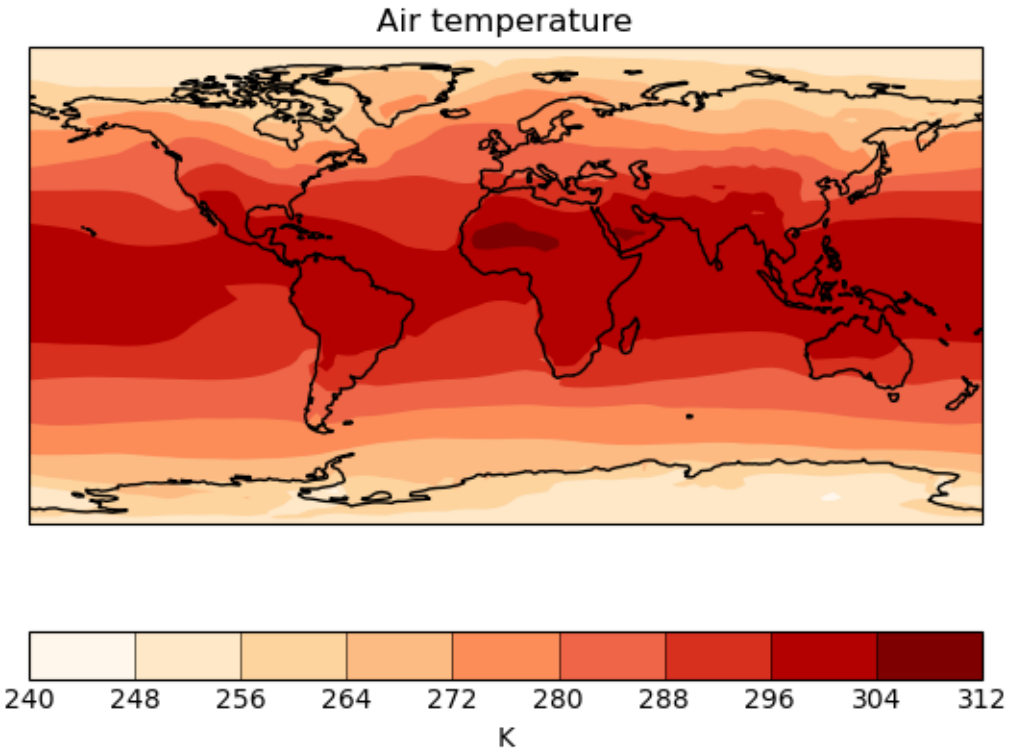
```
import matplotlib.cm as mpl_cm
import matplotlib.pyplot as plt

import iris
import iris.quickplot as qplt

fname = iris.sample_data_path("air_temp.pp")
temperature_cube = iris.load_cube(fname)

# Load a Cynthia Brewer palette.
brewer_cmap = mpl_cm.get_cmap("brewer_OrRd_09")

# Draw the contours, with n-levels set for the map colours (9).
# NOTE: needed as the map is non-interpolated, but matplotlib does not provide
# any special behaviour for these.
qplt.contourf(temperature_cube, brewer_cmap.N, cmap=brewer_cmap)

# Add coastlines to the map created by contourf.
```

(continues on next page)


brewer_Accent_08
brewer_Blues_09
brewer_BrBG_11
brewer_BuGn_09
brewer_BuPu_09
brewer_Dark2_08
brewer_GnBu_09
brewer_Greens_09
brewer_Greys_09
brewer_OrRd_09
brewer_Oranges_09
brewer_PRGn_11
brewer_Paired_12
brewer_Pastel1_09
brewer_Pastel2_08
brewer_PiYG_11
brewer_PuBuGn_09
brewer_PuBu_09
brewer_PuOr_11
brewer_PuRd_09
brewer_Purples_09
brewer_RdBu_11
brewer_RdGy_11
brewer_RdPu_09
brewer_RdYlBu_11
brewer_RdYlGn_11
brewer_Reds_09
brewer_Set1_09
brewer_Set2_08
brewer_Set3_12
brewer_Spectral_11
brewer_YlGnBu_09
brewer_YlGn_09
brewer_YlOrBr_09
brewer_YlOrRd_09

(continued from previous page)

```
plt.gca().coastlines()

plt.show()
```



10.3.3 Adding a Citation

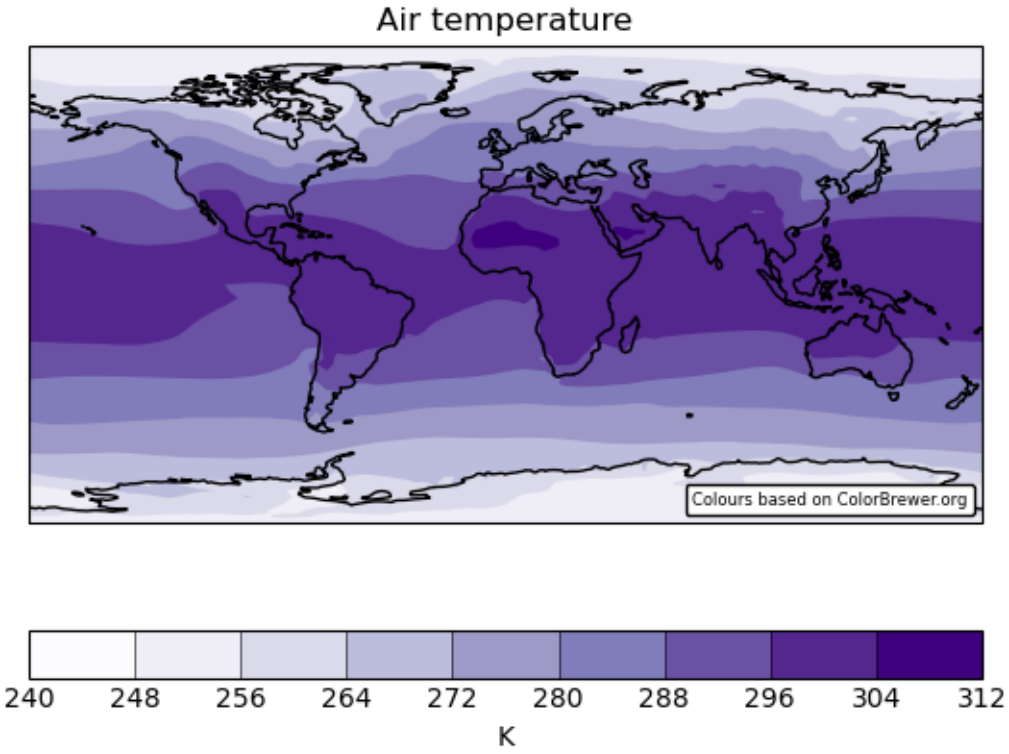
Citations can be easily added to a plot using the *iris.plot.citation()* function. The recommended text for the Cynthia Brewer citation is provided by `iris.plot.BREWER_CITE`.

```
import matplotlib.pyplot as plt

import iris
import iris.plot as iplt
import iris.quickplot as qplt

fname = iris.sample_data_path("air_temp.pp")
temperature_cube = iris.load_cube(fname)

# Get the Purples "Brewer" palette.
brewer_cmap = plt.get_cmap("brewer_Purples_09")
```

(continues on next page)

(continued from previous page)

```
# Draw the contours, with n-levels set for the map colours (9).
# NOTE: needed as the map is non-interpolated, but matplotlib does not provide
# any special behaviour for these.
qplt.contourf(temperature_cube, brewer_cmap.N, cmap=brewer_cmap)

# Add a citation to the plot.
iplt.citation(iris.plot.BREWER_CITE)

# Add coastlines to the map created by contourf.
plt.gca().coastlines()

plt.show()
```



CHAPTER

ELEVEN

CUBE INTERPOLATION AND REGRIDDING

Iris provides powerful cube-aware interpolation and regridding functionality, exposed through Iris cube methods. This functionality is provided by building upon existing interpolation schemes implemented by SciPy.

In Iris we refer to the available types of interpolation and regridding as *schemes*. The following are the interpolation schemes that are currently available in Iris:

- linear interpolation (*iris.analysis.Linear*), and
- nearest-neighbour interpolation (*iris.analysis.Nearest*).

The following are the regridding schemes that are currently available in Iris:

- linear regridding (*iris.analysis.Linear*),
- nearest-neighbour regridding (*iris.analysis.Nearest*), and
- area-weighted regridding (*iris.analysis.AreaWeighted*, first-order conservative).

The linear, nearest-neighbor, and area-weighted regridding schemes support lazy regridding, i.e. if the source cube has lazy data, the resulting cube will also have lazy data. See *Real and Lazy Data* for an introduction to lazy data.

11.1 Interpolation

Interpolating a cube is achieved with the *interpolate()* method. This method expects two arguments:

1. the sample points to interpolate, and
2. the second argument being the interpolation scheme to use.

The result is a new cube, interpolated at the sample points.

Sample points must be defined as an iterable of `(coord, value(s))` pairs. The *coord* argument can be either a coordinate name or coordinate instance. The specified coordinate must exist on the cube being interpolated! For example:

- coordinate names and scalar sample points: `[('latitude', 51.48), ('longitude', 0)]`,
- a coordinate instance and a scalar sample point: `[(cube.coord('latitude'), 51.48)]`, and
- a coordinate name and a NumPy array of sample points: `[('longitude', np.linspace(-11, 2, 14))]`

are all examples of valid sample points.

The values for coordinates that correspond to date/times can be supplied as datetime.datetime or cftime.datetime instances, e.g. `[('time', datetime.datetime(2009, 11, 19, 10, 30))]`).

Let's take the air temperature cube we've seen previously:

```
>>> air_temp = iris.load_cube(iris.sample_data_path('air_temp.pp'))
>>> print(air_temp)
air_temperature / (K)               (latitude: 73; longitude: 96)
    Dimension coordinates:
        latitude                           x              -
        longitude                          -              x
    Scalar coordinates:
        forecast_period             6477 hours, bound=(-28083.0, 6477.0) hours
        forecast_reference_time     1998-03-01 03:00:00
        pressure                    1000.0 hPa
        time                        1998-12-01 00:00:00, bound=(1994-12-01 00:00:00,␣
↪1998-12-01 00:00:00)
    Cell methods:
        mean within years           time
        mean over years             time
    Attributes:
        STASH                       m01s16i203
        source                      Data from Met Office Unified Model
```

We can interpolate specific values from the coordinates of the cube:

```
>>> sample_points = [('latitude', 51.48), ('longitude', 0)]
>>> print(air_temp.interpolate(sample_points, iris.analysis.Linear()))
air_temperature / (K)               (scalar cube)
    Scalar coordinates:
        forecast_period             6477 hours, bound=(-28083.0, 6477.0) hours
        forecast_reference_time     1998-03-01 03:00:00
        latitude                    51.48 degrees
        longitude                   0 degrees
        pressure                    1000.0 hPa
        time                        1998-12-01 00:00:00, bound=(1994-12-01 00:00:00,␣
↪1998-12-01 00:00:00)
    Cell methods:
        mean within years           time
        mean over years             time
    Attributes:
        STASH                       m01s16i203
        source                      Data from Met Office Unified Model
```

As we can see, the resulting cube is scalar and has longitude and latitude coordinates with the values defined in our sample points.

It isn't necessary to specify sample points for every dimension, only those that you wish to interpolate over:

```
>>> result = air_temp.interpolate([('longitude', 0)], iris.analysis.Linear())
>>> print('Original: ' + air_temp.summary(shorten=True))
Original: air_temperature / (K)               (latitude: 73; longitude: 96)
>>> print('Interpolated: ' + result.summary(shorten=True))
Interpolated: air_temperature / (K)               (latitude: 73)
```

The sample points for a coordinate can be an array of values. When multiple coordinates are provided with arrays instead of scalar sample points, the coordinates on the resulting cube will be orthogonal:

```
>>> sample_points = [('longitude', np.linspace(-11, 2, 14)),
...                  ('latitude',  np.linspace(48, 60, 13))]
>>> result = air_temp.interpolate(sample_points, iris.analysis.Linear())
>>> print(result.summary(shorten=True))
air_temperature / (K)               (latitude: 13; longitude: 14)
```

11.1.1 Interpolating Non-Horizontal Coordinates

Interpolation in Iris is not limited to horizontal-spatial coordinates - any coordinate satisfying the prerequisites of the chosen scheme may be interpolated over.

For instance, the *iris.analysis.Linear* scheme requires 1D numeric, monotonic, coordinates. Supposing we have a single column cube such as the one defined below:

```
>>> cube = iris.load_cube(iris.sample_data_path('hybrid_height.nc'), 'air_potential_
↪temperature')
>>> column = cube[:, 0, 0]
>>> print(column.summary(shorten=True))
air_potential_temperature / (K)     (model_level_number: 15)
```

This cube has a "hybrid-height" vertical coordinate system, meaning that the vertical coordinate is unevenly spaced in altitude:

```
>>> print(column.coord('altitude').points)
[ 418.69836  434.5705   456.7928   485.3665   520.2933   561.5752
  609.2145   663.2141   723.57697  790.30664  863.4072   942.8823
 1028.737   1120.9764  1219.6051 ]
```

We could regularise the vertical coordinate by defining 10 equally spaced altitude sample points between 400 and 1250 and interpolating our vertical coordinate onto these sample points:

```
>>> sample_points = [('altitude', np.linspace(400, 1250, 10))]
>>> new_column = column.interpolate(sample_points, iris.analysis.Linear())
>>> print(new_column.summary(shorten=True))
air_potential_temperature / (K)     (model_level_number: 10)
```

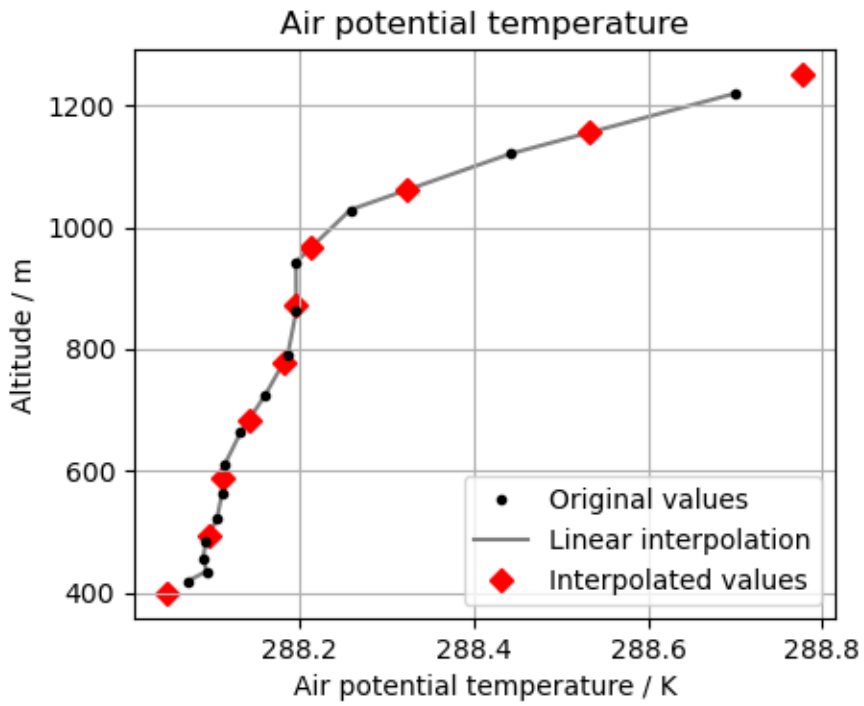
Let's look at the original data, the interpolation line and the new data in a plot. This will help us to see what is going on:

The red diamonds on the extremes of the altitude values show that we have extrapolated data beyond the range of the original data. In some cases this is desirable but in other cases it is not. For example, this column defines a surface altitude value of 414m, so extrapolating an "air potential temperature" at 400m makes little physical sense in this case.

We can control the extrapolation mode when defining the interpolation scheme. Controlling the extrapolation mode allows us to avoid situations like the above where extrapolating values makes little physical sense.

The extrapolation mode is controlled by the `extrapolation_mode` keyword. For the available interpolation schemes available in Iris, the `extrapolation_mode` keyword must be one of:

- `extrapolate` – the extrapolation points will be calculated by extending the gradient of the closest two points,
- `error` – a ValueError exception will be raised, notifying an attempt to extrapolate,
- `nan` – the extrapolation points will be be set to NaN,
- `mask` – the extrapolation points will always be masked, even if the source data is not a MaskedArray, or



- `nanmask` – if the source data is a MaskedArray the extrapolation points will be masked. Otherwise they will be set to NaN.

Using an extrapolation mode is achieved by constructing an interpolation scheme with the extrapolation mode keyword set as required. The constructed scheme is then passed to the *`interpolate()`* method. For example, to mask values that lie beyond the range of the original data:

```
>>> scheme = iris.analysis.Linear(extrapolation_mode='mask')
>>> new_column = column.interpolate(sample_points, scheme)
>>> print(new_column.coord('altitude').points)
[-- 494.44451904296875 588.888916015625 683.333251953125 777.77783203125
 872.2222290039062 966.666748046875 1061.111083984375 1155.555419921875 --]
```

11.1.2 Caching an Interpolator

If you need to interpolate a cube on multiple sets of sample points you can 'cache' an interpolator to be used for each of these interpolations. This can shorten the execution time of your code as the most computationally intensive part of an interpolation is setting up the interpolator.

To cache an interpolator you must set up an interpolator scheme and call the scheme's interpolator method. The interpolator method takes as arguments:

1. a cube to be interpolated, and
2. an iterable of coordinate names or coordinate instances of the coordinates that are to be interpolated over.

For example:

```
>>> air_temp = iris.load_cube(iris.sample_data_path('air_temp.pp'))
>>> interpolator = iris.analysis.Nearest().interpolator(air_temp, ['latitude', 'longitude
↪'])
```

When this cached interpolator is called you must pass it an iterable of sample points that have the same form as the iterable of coordinates passed to the constructor. So, to use the cached interpolator defined above:

```
>>> latitudes = np.linspace(48, 60, 13)
>>> longitudes = np.linspace(-11, 2, 14)
>>> for lat, lon in zip(latitudes, longitudes):
...     result = interpolator([lat, lon])
```

In each case `result` will be a cube interpolated from the `air_temp` cube we passed to interpolator.

Note that you must specify the required extrapolation mode when setting up the cached interpolator. For example:

```
>>> interpolator = iris.analysis.Nearest(extrapolation_mode='nan').interpolator(cube,␣
↪coords)
```

11.2 Regridding

Regridding is conceptually a very similar process to interpolation in Iris. The primary difference is that interpolation is based on sample points, while regridding is based on the **horizontal** grid of *another cube*.

Regridding a cube is achieved with the *cube.regrid()* method. This method expects two arguments:

1. *another cube* that defines the target grid onto which the cube should be regridded, and
2. the regridding scheme to use.

Note: Regridding is a common operation needed to allow comparisons of data on different grids. The powerful mapping functionality provided by cartopy, however, means that regridding is often not necessary if performed just for visualisation purposes.

Let's load two cubes that have different grids and coordinate systems:

```
>>> global_air_temp = iris.load_cube(iris.sample_data_path('air_temp.pp'))
>>> rotated_psl = iris.load_cube(iris.sample_data_path('rotated_pole.nc'))
```

We can visually confirm that they are on different grids by plotting the two cubes:

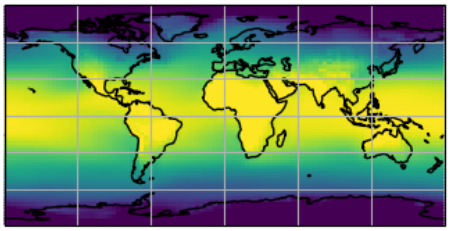
Let's regrid the `global_air_temp` cube onto a rotated pole grid using a linear regridding scheme. To achieve this we pass the `rotated_psl` cube to the regridder to supply the target grid to regrid the `global_air_temp` cube onto:

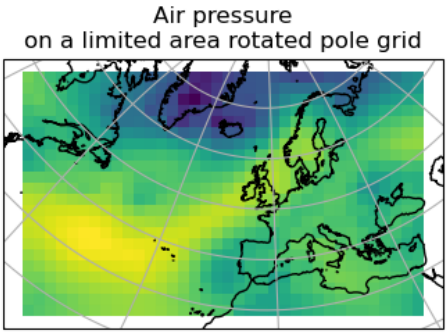
```
>>> rotated_air_temp = global_air_temp.regrid(rotated_psl, iris.analysis.Linear())
```

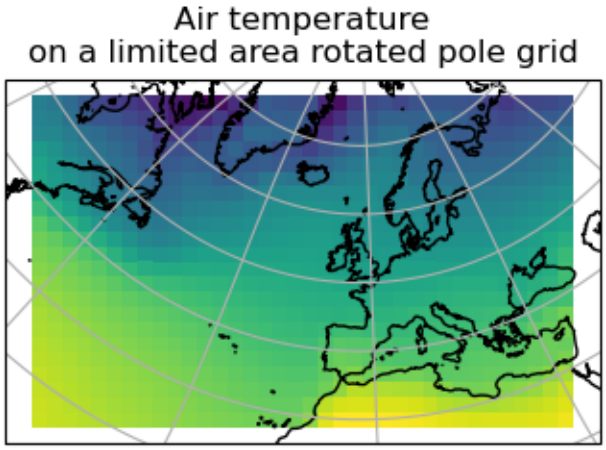
We could regrid the pressure values onto the global grid, but this will involve some form of extrapolation. As with interpolation, we can control the extrapolation mode when defining the regridding scheme.

For the available regridding schemes in Iris, the `extrapolation_mode` keyword must be one of:

- `extrapolate` –
  - for *Linear* the extrapolation points will be calculated by extending the gradient of the closest two points.
  - for *Nearest* the extrapolation points will take their value from the nearest source point.


Air temperature
on a global longitude latitude grid

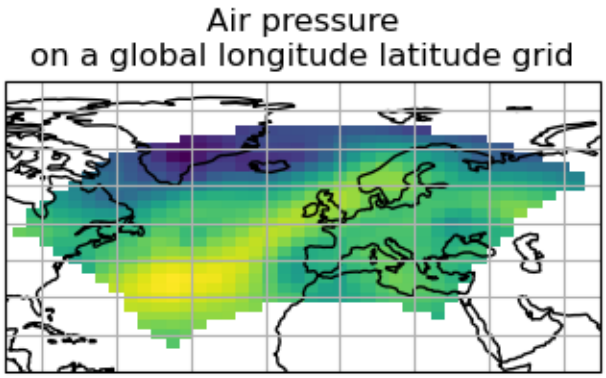

Air pressure
on a limited area rotated pole grid


Air temperature
on a limited area rotated pole grid

- `nan` – the extrapolation points will be be set to NaN.
- `error` – a ValueError exception will be raised, notifying an attempt to extrapolate.
- `mask` – the extrapolation points will always be masked, even if the source data is not a MaskedArray.
- `nanmask` – if the source data is a MaskedArray the extrapolation points will be masked. Otherwise they will be set to NaN.

The `rotated_psl` cube is defined on a limited area rotated pole grid. If we regridded the `rotated_psl` cube onto the global grid as defined by the `global_air_temp` cube any linearly extrapolated values would quickly become dominant and highly inaccurate. We can control this behaviour by defining the `extrapolation_mode` in the constructor of the regridding scheme to mask values that lie outside of the domain of the rotated pole grid:

```
>>> scheme = iris.analysis.Linear(extrapolation_mode='mask')
>>> global_psl = rotated_psl.regrid(global_air_temp, scheme)
```



Notice that although we can still see the approximate shape of the rotated pole grid, the cells have now become rectangular in a plate carrée (equirectangular) projection. The spatial grid of the resulting cube is really global, with a large proportion of the data being masked.

11.2.1 Area-Weighted Regridding

It is often the case that a point-based regridding scheme (such as *iris.analysis.Linear* or *iris.analysis.Nearest*) is not appropriate when you need to conserve quantities when regridding. The *iris.analysis.AreaWeighted* scheme is less general than *Linear* or *Nearest*, but is a conservative regridding scheme, meaning that the area-weighted total is approximately preserved across grids.

With the *AreaWeighted* regridding scheme, each target grid-box's data is computed as a weighted mean of all grid-boxes from the source grid. The weighting for any given target grid-box is the area of the intersection with each of the source grid-boxes. This scheme performs well when regridding from a high resolution source grid to a lower resolution target grid, since all source data points will be accounted for in the target grid.

Let's demonstrate this with the global air temperature cube we saw previously, along with a limited area cube containing total concentration of volcanic ash:

```
>>> global_air_temp = iris.load_cube(iris.sample_data_path('air_temp.pp'))
>>> print(global_air_temp.summary(shorten=True))
air_temperature / (K)               (latitude: 73; longitude: 96)
>>>
>>> regional_ash = iris.load_cube(iris.sample_data_path('NAME_output.txt'))
>>> regional_ash = regional_ash.collapsed('flight_level', iris.analysis.SUM)
>>> print(regional_ash.summary(shorten=True))
VOLCANIC_ASH_AIR_CONCENTRATION / (g/m3) (latitude: 214; longitude: 584)
```

One of the key limitations of the *AreaWeighted* regridding scheme is that the two input grids must be defined in the same coordinate system as each other. Both input grids must also contain monotonic, bounded, 1D spatial coordinates.

Note: The *AreaWeighted* regridding scheme requires spatial areas, therefore the longitude and latitude coordinates must be bounded. If the longitude and latitude bounds are not defined in the cube we can guess the bounds based on the coordinates' point values:

```
>>> global_air_temp.coord('longitude').guess_bounds()
>>> global_air_temp.coord('latitude').guess_bounds()
```

Using NumPy's masked array module we can mask any data that falls below a meaningful concentration:

```
>>> regional_ash.data = np.ma.masked_less(regional_ash.data, 5e-6)
```

Finally, we can regrid the data using the *AreaWeighted* regridding scheme:

```
>>> scheme = iris.analysis.AreaWeighted(mdtol=0.5)
>>> global_ash = regional_ash.regrid(global_air_temp, scheme)
>>> print(global_ash.summary(shorten=True))
VOLCANIC_ASH_AIR_CONCENTRATION / (g/m3) (latitude: 73; longitude: 96)
```

Note that the *AreaWeighted* regridding scheme allows us to define a missing data tolerance (`mdtol`), which specifies the tolerated fraction of masked data in any given target grid-box. If the fraction of masked data within a target grid-box exceeds this value, the data in this target grid-box will be masked in the result.

The fraction of masked data is calculated based on the area of masked source grid-boxes that overlaps with each target grid-box. Defining an `mdtol` in the *AreaWeighted* regridding scheme allows fine control of masked data tolerance. It is worth remembering that defining an `mdtol` of anything other than 1 will prevent the scheme from being fully conservative, as some data will be disregarded if it lies close to masked data.

To visualise the above regrid, let's plot the original data, along with 3 distinct `mdtol` values to compare the result:

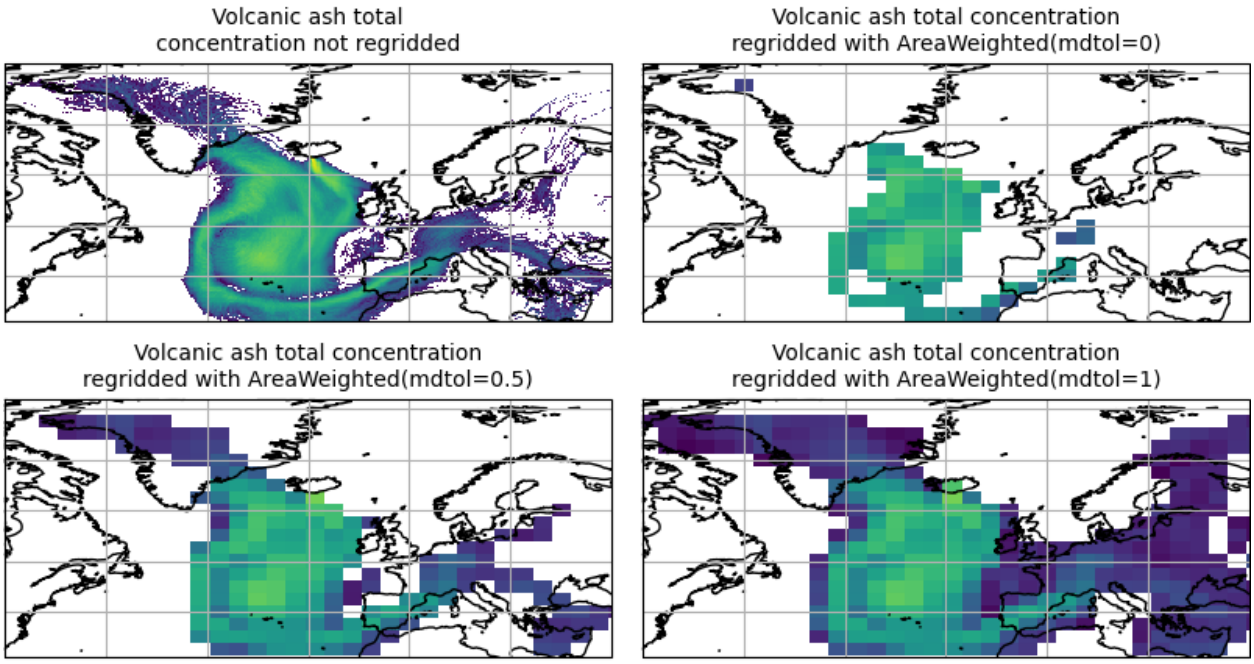
11.2.2 Caching a Regridder

If you need to regrid multiple cubes with a common source grid onto a common target grid you can 'cache' a regridder to be used for each of these regrids. This can shorten the execution time of your code as the most computationally intensive part of a regrid is setting up the regridder.

To cache a regridder you must set up a regridder scheme and call the scheme's regridder method. The regridder method takes as arguments:

1. a cube (that is to be regridded) defining the source grid, and
2. a cube defining the target grid to regrid the source cube to.

For example:




```
>>> global_air_temp = iris.load_cube(iris.sample_data_path('air_temp.pp'))
>>> rotated_psl = iris.load_cube(iris.sample_data_path('rotated_pole.nc'))
>>> regridder = iris.analysis.Nearest().regridder(global_air_temp, rotated_psl)
```

When this cached regridder is called you must pass it a cube on the same grid as the source grid cube (in this case `global_air_temp`) that is to be regridded to the target grid. For example:

```
>>> for cube in list_of_cubes_on_source_grid:
...     result = regridder(cube)
```

In each case `result` will be the input cube regridded to the grid defined by the target grid cube (in this case `rotated_psl`) that we used to define the cached regridder.

11.2.3 Regridding Lazy Data

If you are working with large cubes, especially when you are regridding to a high resolution target grid, you may run out of memory when trying to regrid a cube. When this happens, make sure the input cube has lazy data

```
>>> air_temp = iris.load_cube(iris.sample_data_path('A1B_north_america.nc'))
>>> air_temp
<iris 'Cube' of air_temperature / (K) (time: 240; latitude: 37; longitude: 49)>
>>> air_temp.has_lazy_data()
True
```

and the regridding scheme supports lazy data. All regridding schemes described here support lazy data. If you still run out of memory even while using lazy data, inspect the chunks :

```
>>> air_temp.lazy_data().chunks
((240,), (37,), (49,))
```

The cube above consist of a single chunk, because it is fairly small. For larger cubes, iris will automatically create chunks of an optimal size when loading the data. However, because regridding to a high resolution grid may dramatically increase the size of the data, the automatically chosen chunks might be too large.

As an example of how to solve this, we could manually re-chunk the time dimension, to regrid it in 8 chunks of 30 timesteps at a time:

```
>>> air_temp.data = air_temp.lazy_data().rechunk([30, None, None])
>>> air_temp.lazy_data().chunks
((30, 30, 30, 30, 30, 30, 30, 30), (37,), (49,))
```

Assuming that Dask is configured such that it processes only a few chunks of the data array at a time, this will further reduce memory use.

Note that chunking in the horizontal dimensions is not supported by the regridding schemes. Chunks in these dimensions will automatically be combined before regridding.

CHAPTER

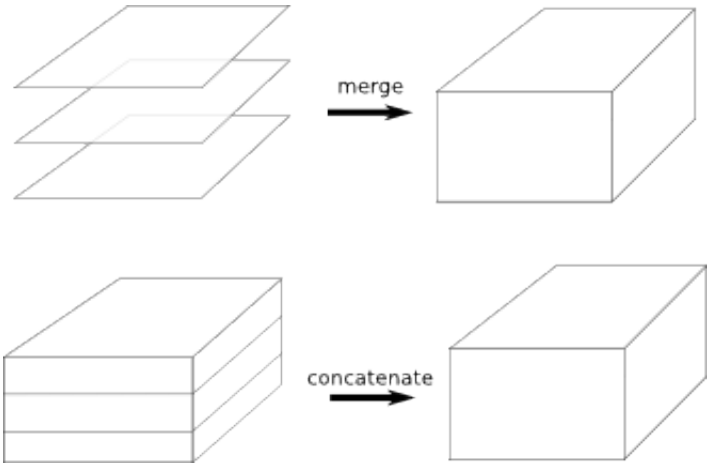
TWELVE

MERGE AND CONCATENATE

We saw in the *Loading Iris Cubes* chapter that Iris tries to load as few cubes as possible. This is done by collecting together multiple fields with a shared standard name (and other key metadata) into a single multidimensional cube. The processes that perform this behaviour in Iris are known as `merge` and `concatenate`.

This chapter describes the `merge` and `concatenate` processes; it explains why common issues occur when using them and gives advice on how prevent these issues from occurring.

Both `merge` and `concatenate` take multiple cubes as input and result in fewer cubes as output. The following diagram illustrates the two processes:



There is one major difference between the `merge` and `concatenate` processes.

- The `merge` process combines multiple input cubes into a single resultant cube with new dimensions created from the *scalar coordinate values* of the input cubes.
- The `concatenate` process combines multiple input cubes into a single resultant cube with the same *number of dimensions* as the input cubes, but with the length of one or more dimensions extended by *joining together sequential dimension coordinates*.

Let's imagine 28 individual cubes representing the temperature at a location `(y, x)`; one cube for each day of February. We can use *`merge()`* to combine the 28 `(y, x)` cubes into a single `(t, y, x)` cube, where the length of the `t` dimension is 28.

Now imagine 12 individual cubes representing daily temperature at a time and location `(t, y, x)`; one cube for each month in the year. We can use *`concatenate()`* to combine the 12 `(t, y, x)` cubes into a single `(t, y, x)` cube, where the length of the `t` dimension is now 365.

12.1 Merge

We've seen that the `merge` process combines multiple input cubes into a single resultant cube with new dimensions created from the *scalar coordinate values* of the input cubes.

In order to construct new coordinates for the new dimensions, the `merge` process requires input cubes with scalar coordinates that can be combined together into monotonic sequences. The order of the input cubes does not affect the `merge` process.

The `merge` process can produce a cube that has more than one new dimension, if the scalar coordinate sequences form an orthogonal basis.

Important: The shape, metadata, attributes, coordinates, coordinates metadata, fill value and other aspects of the input cubes must be consistent across all of the input cubes.

The `merge` process will fail if these are not consistent. Such failures are covered in the *Common Issues With Merge and Concatenate* section.

The `merge` process can be accessed using two methods. The two methods are *merge()* and *merge_cube()*, which are described below.

12.1.1 Using CubeList.merge

The *CubeList.merge* method operates on a list of cubes and returns a new *CubeList* containing the cubes that have been merged.

Let's have a look at the *merge()* method in operation. In this example we have a list of three lateral (x, y) cubes in a variable called `cubes`, each with a scalar `z` coordinate of differing value. We can merge these cubes by stacking the scalar `z` coordinates to make a new `z` dimension coordinate:

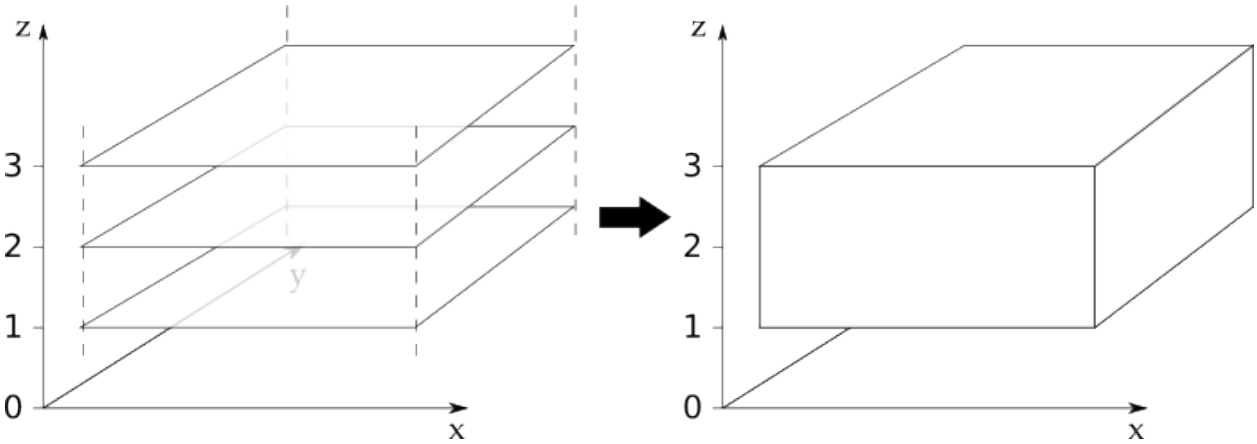
```
>>> print(cubes)
0: air_temperature / (kelvin)          (y: 4; x: 5)
1: air_temperature / (kelvin)          (y: 4; x: 5)
2: air_temperature / (kelvin)          (y: 4; x: 5)

>>> print(cubes[0])
air_temperature / (kelvin)          (y: 4; x: 5)
 ...
    Scalar coordinates:
        z                           1 meters
>>> print(cubes[1])
air_temperature / (kelvin)          (y: 4; x: 5)
 ...
    Scalar coordinates:
        z                           2 meters
>>> print(cubes[2])
air_temperature / (kelvin)          (y: 4; x: 5)
    Dimension coordinates:
        y                             x     -
        x                             -     x
    Scalar coordinates:
        z                           3 meters
```

(continues on next page)

(continued from previous page)

```
>>> print(cubes.merge())
0: air_temperature / (kelvin)          (z: 3; y: 4; x: 5)
```

The following diagram illustrates what has taken place in this example:



The diagram illustrates that we have three input cubes of identical shape that stack on the `z` dimension. After merging our three input cubes we get a new *CubeList* containing one cube with a new `z` coordinate.

12.1.2 Using CubeList.merge_cube

The *merge_cube()* method guarantees that *exactly one cube will be returned* as a result of merging the input cubes. If *merge_cube()* cannot fulfil this guarantee, a descriptive error will be raised providing details to help diagnose the differences between the input cubes. In contrast, the *merge()* method makes no check on the number of cubes returned.

To demonstrate the differences between *merge()* and *merge_cube()*, let's return to our three cubes from the earlier merge example.

For the purposes of this example a `Conventions` attribute has been added to the first cube's *attributes* dictionary. Remember that the attributes *must* be consistent across all cubes in order to merge into a single cube:

```
>>> print(cubes)
0: air_temperature / (kelvin)          (y: 4; x: 5)
1: air_temperature / (kelvin)          (y: 4; x: 5)
2: air_temperature / (kelvin)          (y: 4; x: 5)

>>> print(cubes[0].attributes)
{'Conventions': 'CF-1.5'}
>>> print(cubes[1].attributes)
{}
>>> print(cubes[2].attributes)
{}

>>> print(cubes.merge())
0: air_temperature / (kelvin)          (y: 4; x: 5)
1: air_temperature / (kelvin)          (z: 2; y: 4; x: 5)

>>> print(cubes.merge_cube())
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
    ...
    raise iris.exceptions.MergeError(msgs)
iris.exceptions.MergeError: failed to merge into a single cube.
  cube.attributes keys differ: 'Conventions'
```

Note that *merge()* returns two cubes here. All the cubes that can be merged have been merged. Any cubes that can't be merged are included unchanged in the returned *CubeList*. When *merge_cube()* is called on `cubes` it raises a descriptive error that highlights the difference in the `attributes` dictionaries. It is this difference that is preventing `cubes` being merged into a single cube. An example of fixing an issue like this can be found in the *Common Issues With Merge and Concatenate* section.

12.1.3 Merge in Iris Load

The CubeList's *merge()* method is used internally by the three main Iris load functions introduced in *Loading Iris Cubes*. For file formats such as GRIB and PP, which store fields as many individual 2D arrays, Iris loading uses the `merge` process to produce a more intuitive higher dimensional cube of each phenomenon where possible.

Sometimes the `merge` process doesn't behave as expected. In almost all cases this is due to the input cubes containing unexpected or inconsistent metadata. For this reason, a fourth Iris file loading function, *iris.load_raw()*, exists. The *load_raw()* function is intended as a diagnostic tool that can be used to load cubes from files without the `merge` process taking place. The return value of *iris.load_raw()* is always a *CubeList* instance. You can then call the *merge_cube()* method on this returned *CubeList* to help identify merge related load issues.

12.2 Concatenate

We've seen that the `concatenate` process combines multiple input cubes into a single resultant cube with the same *number of dimensions* as the input cubes, but with the length of one or more dimensions extended by *joining together sequential dimension coordinates*.

In order to extend the dimensions lengths, the `concatenate` process requires input cubes with dimension coordinates that can be combined together into monotonic sequences. The order of the input cubes does not affect the `concatenate` process.

Important: The shape, metadata, attributes, coordinates, coordinates metadata, fill value and other aspects of the input cubes must be consistent across all of the input cubes.

The `concatenate` process will fail if these are not consistent. Such failures are covered in the *Common Issues With Merge and Concatenate* section.

The `concatenate` process can be accessed using two methods. The two methods are *concatenate()* and *concatenate_cube()*, which are described below.

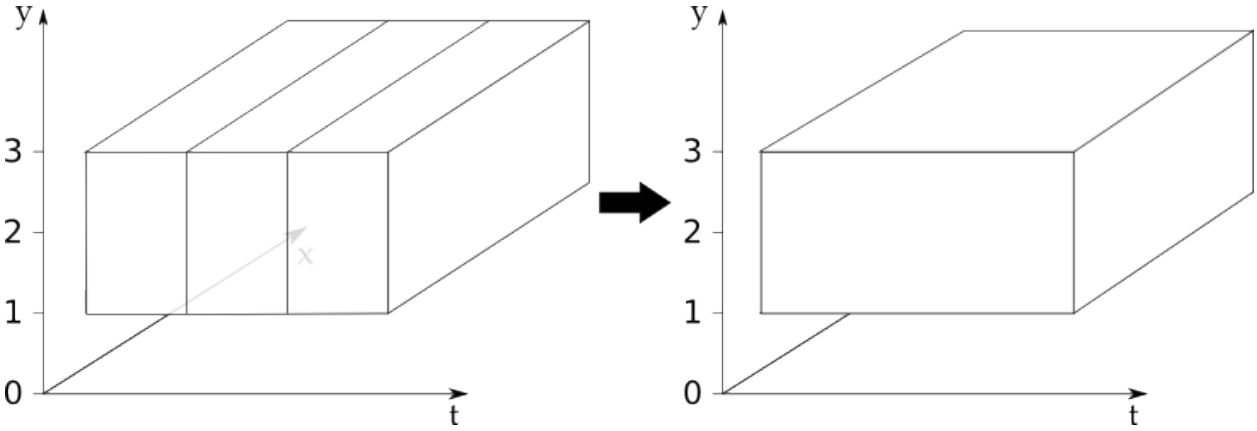
12.2.1 Using CubeList.concatenate

The *CubeList.concatenate* method operates on a list of cubes and returns a new *CubeList* containing the cubes that have been concatenated.

Let's have a look at the *concatenate()* method in operation. In the example below we have three 3D `(t, y, x)` cubes whose `t` coordinates have sequentially increasing ranges. These cubes can be concatenated by combining the `t` coordinates of the input cubes to form a new cube with an extended `t` coordinate:

```
>>> print(cubes)
0: air_temperature / (kelvin)          (t: 31; y: 3; x: 4)
1: air_temperature / (kelvin)          (t: 28; y: 3; x: 4)
2: air_temperature / (kelvin)          (t: 31; y: 3; x: 4)

>>> print(cubes.concatenate())
0: air_temperature / (kelvin)          (t: 90; y: 3; x: 4)
```

The following diagram illustrates what has taken place in this example:



The diagram illustrates that we have three 3D input cubes that line up on the `t` dimension. After concatenating our three input cubes we get a new *CubeList* containing one cube with an extended `t` coordinate.

12.2.2 Using CubeList.concatenate_cube

The *concatenate_cube()* method guarantees that *exactly one cube will be returned* as a result of concatenating the input cubes. If *concatenate_cube()* cannot fulfil this guarantee, a descriptive error will be raised providing details to help diagnose the differences between the input cubes. In contrast, the *concatenate()* method makes no check on the number of cubes returned.

To demonstrate the differences between *concatenate()* and *concatenate_cube()*, let's return to our three cubes from the earlier concatenate example.

For the purposes of this example we'll add a *History* attribute to the first cube's *attributes* dictionary. Remember that the attributes *must* be consistent across all cubes in order to concatenate into a single cube:

```
>>> print(cubes)
0: air_temperature / (kelvin)          (t: 31; y: 3; x: 4)
1: air_temperature / (kelvin)          (t: 28; y: 3; x: 4)
2: air_temperature / (kelvin)          (t: 31; y: 3; x: 4)

>>> print(cubes[0].attributes)
```

(continues on next page)

(continued from previous page)

```
{'History': 'Created 2010-06-30'}
>>> print(cubes[1].attributes)
{}

>>> print(cubes.concatenate())
0: air_temperature / (kelvin)          (t: 31; y: 3; x: 4)
1: air_temperature / (kelvin)          (t: 59; y: 3; x: 4)
>>> print(cubes.concatenate_cube())
Traceback (most recent call last):
    ...
    raise iris.exceptions.ConcatenateError(msgs)
iris.exceptions.ConcatenateError: failed to concatenate into a single cube.
  Cube metadata differs for phenomenon: air_temperature
```

Note that *concatenate()* returns two cubes here. All the cubes that can be concatenated have been concatenated. Any cubes that can't be concatenated are included unchanged in the returned *CubeList*. When *concatenate_cube()* is called on `cubes` it raises a descriptive error that highlights the difference in the `attributes` dictionaries. It is this difference that is preventing `cubes` being concatenated into a single cube. An example of fixing an issue like this can be found in the *Common Issues With Merge and Concatenate* section.

12.3 Common Issues With Merge and Concatenate

The Iris algorithms that drive *merge()* and *concatenate()* are complex and depend on a number of different elements of the input cubes being consistent across all input cubes. If this consistency is not maintained then the *merge()* or *concatenate()* process can fail in a seemingly arbitrary manner.

The methods *merge_cube()* and *concatenate_cube()* were introduced to Iris to help you locate differences in input cubes that prevent the input cubes merging or concatenating. Nevertheless, certain difficulties with using *merge()* and *concatenate()* occur frequently. This section describes these common difficulties, why they arise and what you can do to avoid them.

12.3.1 Merge

Attributes Mismatch

Differences in the *attributes* the input cubes probably cause the greatest amount of merge-related difficulties. In recognition of this, Iris has a helper function, *equalise_attributes()*, to equalise attributes differences in the input cubes.

Note: The functionality provided by *iris.util.describe_diff()* and *iris.cube.Cube.is_compatible()* are **not** designed to give user indication of whether two cubes can be merged.

To demonstrate using *equalise_attributes()*, let's return to our non-merging list of input cubes from the merge_cube example from earlier. We'll call *equalise_attributes()* on the input cubes before merging the input cubes using *merge_cube()*:

```
>>> from iris.util import equalise_attributes
>>> print(cubes)
0: air_temperature / (kelvin)          (y: 4; x: 5)
1: air_temperature / (kelvin)          (y: 4; x: 5)
```

(continues on next page)

(continued from previous page)

```
2: air_temperature / (kelvin)          (y: 4; x: 5)

>>> print(cubes[0].attributes)
{'Conventions': 'CF-1.5'}
>>> print(cubes[1].attributes)
{}
>>> print(cubes[2].attributes)
{}

>>> print(cubes.merge_cube())
Traceback (most recent call last):
    ...
    raise iris.exceptions.MergeError(msgs)
iris.exceptions.MergeError: failed to merge into a single cube.
  cube.attributes keys differ: 'Conventions'

>>> equalise_attributes(cubes)

>>> print(cubes[0].attributes)
{}

>>> print(cubes.merge_cube())
air_temperature / (kelvin)          (z: 3; y: 4; x: 5)
     Dimension coordinates:
          z                           x     -     -
          y                           -     x     -
          x                           -     -     x
```

Incomplete Data

Merging input cubes with inconsistent dimension lengths can cause misleading results. This is a common problem when merging cubes generated by different ensemble members in a model run.

The misleading results cause the merged cube to gain an anonymous leading dimension. All the merged coordinates appear as auxiliary coordinates on the anonymous leading dimension. This is shown in the example below:

```
>>> print(cube)
surface_temperature / (K)           (-- : 5494; latitude: 325; longitude: 432)
     Dimension coordinates:
          latitude                      -              x              -
          longitude                     -              -              x
     Auxiliary coordinates:
          forecast_month                x              -              -
          forecast_period               x              -              -
          forecast_reference_time       x              -              -
          realization                   x              -              -
          time                          x              -              -
```

Merging Duplicate Cubes

The Iris merge process does not merge duplicate cubes (two or more identical cubes in the input cubes) by default. This behaviour can be changed by setting the `unique` keyword argument to *merge()* to `False`.

Merging duplicate cubes can cause misleading results. Let's demonstrate these behaviours and misleading results with the following example. In this example we have three input cubes. The first has a scalar `z` coordinate with value 1, the

second has a scalar `z` coordinate with value 2 and the third has a scalar `z` coordinate with value 1. The first and third cubes are thus identical. We will demonstrate the effect of merging the input cubes with `unique=False` (duplicate cubes allowed) and `unique=True` (duplicate cubes not allowed, which is the default behaviour):

```
>>> print(cubes)
0: air_temperature / (kelvin)          (y: 4; x: 5)
1: air_temperature / (kelvin)          (y: 4; x: 5)
2: air_temperature / (kelvin)          (y: 4; x: 5)

>>> print(cubes.merge(unique=False))
0: air_temperature / (kelvin)          (z: 2; y: 4; x: 5)
1: air_temperature / (kelvin)          (z: 2; y: 4; x: 5)

>>> print(cubes.merge())  # unique=True is the default.
Traceback (most recent call last):
  ...
iris.exceptions.DuplicateDataError: failed to merge into a single cube.
  Duplicate 'air_temperature' cube, with scalar coordinates z=Cell(point=1, bound=None)
```

Notice how merging the input cubes with duplicate cubes allowed produces a result with **four** *z* coordinate values. Closer inspection of these two resultant cubes demonstrates that the scalar `z` coordinate with value 2 is found in both cubes.

Trying to merge the input cubes with duplicate cubes not allowed raises an error highlighting the presence of the duplicate cube.

Single Value Coordinates

Coordinates containing only a single value can cause confusion when combining input cubes. Remember:

- The `merge` process combines multiple input cubes into a single resultant cube with new dimensions created from the **scalar** *coordinate values* of the input cubes.
- The `concatenate` process combines multiple input cubes into a single resultant cube with the same *number of dimensions* as the input cubes, but with the length of one or more dimensions extended by *joining together sequential* **dimension** *coordinates*.

In Iris terminology a **scalar** coordinate is a coordinate of length 1 *which does not describe a data dimension*.

Let's look at two example cubes to demonstrate this.

If your cubes are similar to those below (the single value `z` coordinate is not on a dimension) then use *merge()* to combine your cubes:

```
>>> print(cubes[0])
air_temperature / (kelvin)          (y: 4; x: 5)
    Dimension coordinates:
        y                             x     -
        x                             -     x
    Scalar coordinates:
        z                           1 meters
>>> print(cubes[1])
air_temperature / (kelvin)          (y: 4; x: 5)
    Dimension coordinates:
        y                             x     -
        x                             -     x
    Scalar coordinates:
        z                           2 meters
```

If your cubes are similar to those below (the single value z coordinate is associated with a dimension) then use *concatenate()* to combine your cubes:

```
>>> print(cubes)
0: air_temperature / (kelvin)          (z: 1; y: 4; x: 5)
1: air_temperature / (kelvin)          (z: 1; y: 4; x: 5)
```

12.3.2 Concatenate

Time Units

Differences in the units of the time coordinates of the input cubes probably cause the greatest amount of concatenate-related difficulties. In recognition of this, Iris has a helper function, *unify_time_units()*, to apply a common time unit to all the input cubes.

To demonstrate using *unify_time_units()*, let's adapt our list of input cubes from the concatenate_cube example from earlier. We'll give the input cubes unequal time coordinate units and call *unify_time_units()* on the input cubes before concatenating the input cubes using *concatenate_cube()*:

```
>>> from iris.util import unify_time_units
>>> print(cubes)
0: air_temperature / (kelvin)          (t: 31; y: 3; x: 4)
1: air_temperature / (kelvin)          (t: 28; y: 3; x: 4)
2: air_temperature / (kelvin)          (t: 31; y: 3; x: 4)

>>> print(cubes[0].coord('t').units)
days since 1990-02-15
>>> print(cubes[1].coord('t').units)
days since 1970-01-01

>>> print(cubes.concatenate_cube())
Traceback (most recent call last):
 ...
ConcatenateError: failed to concatenate into a single cube.
  Dimension coordinates metadata differ: t != t

>>> unify_time_units(cubes)

>>> print(cubes[1].coord('t').units)
days since 1990-02-15

>>> print(cubes.concatenate_cube())
air_temperature / (kelvin)          (t: 90; y: 3; x: 4)
     Dimension coordinates:
          t                           x      -      -
          y                           -      x      -
          x                           -      -      x
```

Attributes Mismatch

The concatenate process is affected by attributes mismatch on input cubes in the same way that the merge process is. The *Attributes Mismatch* section earlier in this chapter gives further information on attributes mismatch.

CHAPTER

THIRTEEN

CUBE STATISTICS

13.1 Collapsing Entire Data Dimensions

In the *Subsetting a Cube* section we saw how to extract a subset of a cube in order to reduce either its dimensionality or its resolution. Instead of simply extracting a sub-region of the data, we can produce statistical functions of the data values across a particular dimension, such as a 'mean over time' or 'minimum over latitude'.

For instance, suppose we have a cube:

```
>>> import iris
>>> filename = iris.sample_data_path('uk_hires.pp')
>>> cube = iris.load_cube(filename, 'air_potential_temperature')
>>> print(cube)
air_potential_temperature / (K)     (time: 3; model_level_number: 7; grid_latitude: 204;␣
↪grid_longitude: 187)
    Dimension coordinates:
        time                             x                      -                 -      ␣
↪               -
        model_level_number               -                      x                 -      ␣
↪               -
        grid_latitude                    -                      -                 x      ␣
↪               -
        grid_longitude                   -                      -                 -      ␣
↪               x
    Auxiliary coordinates:
        forecast_period                  x                      -                 -      ␣
↪               -
        level_height                     -                      x                 -      ␣
↪               -
        sigma                            -                      x                 -      ␣
↪               -
        surface_altitude                 -                      -                 x      ␣
↪               x
    Derived coordinates:
        altitude                         -                      x                 x      ␣
↪               x
    Scalar coordinates:
        forecast_reference_time     2009-11-19 04:00:00
    Attributes:
        STASH                       m01s00i004
```

(continues on next page)

(continued from previous page)

```
        source                      Data from Met Office Unified Model
        um_version                  7.3
```

In this case we have a 4 dimensional cube; to mean the vertical (z) dimension down to a single valued extent we can pass the coordinate name and the aggregation definition to the *Cube.collapsed()* method:

```
>>> import iris.analysis
>>> vertical_mean = cube.collapsed('model_level_number', iris.analysis.MEAN)
>>> print(vertical_mean)
air_potential_temperature / (K)     (time: 3; grid_latitude: 204; grid_longitude: 187)
    Dimension coordinates:
        time                            x                 -                   -
        grid_latitude                   -                 x                   -
        grid_longitude                  -                 -                   x
    Auxiliary coordinates:
        forecast_period                 x                 -                   -
        surface_altitude                -                 x                   x
    Derived coordinates:
        altitude                        -                 x                   x
    Scalar coordinates:
        forecast_reference_time     2009-11-19 04:00:00
        level_height                696.6666 m, bound=(0.0, 1393.3333) m
        model_level_number          10, bound=(1, 19)
        sigma                       0.92292976, bound=(0.8458596, 1.0)
    Cell methods:
        mean                        model_level_number
    Attributes:
        STASH                       m01s00i004
        source                      Data from Met Office Unified Model
        um_version                  7.3
```

Similarly other analysis operators such as `MAX`, `MIN` and `STD_DEV` can be used instead of `MEAN`, see *iris.analysis* for a full list of currently supported operators.

For an example of using this functionality, the *Hovmoller Diagram of Monthly Surface Temperature* example found in the gallery takes a zonal mean of an `XYT` cube by using the `collapsed` method with `latitude` and `iris.analysis.MEAN` as arguments.

13.1.1 Area Averaging

Some operators support additional keywords to the `cube.collapsed` method. For example, *iris.analysis.MEAN* supports a weights keyword which can be combined with *iris.analysis.cartography.area_weights()* to calculate an area average.

Let's use the same data as was loaded in the previous example. Since `grid_latitude` and `grid_longitude` were both point coordinates we must guess bound positions for them in order to calculate the area of the grid boxes:

```
import iris.analysis.cartography
cube.coord('grid_latitude').guess_bounds()
cube.coord('grid_longitude').guess_bounds()
grid_areas = iris.analysis.cartography.area_weights(cube)
```

These areas can now be passed to the `collapsed` method as weights:

```
>>> new_cube = cube.collapsed(['grid_longitude', 'grid_latitude'], iris.analysis.MEAN,␣
↪weights=grid_areas)
>>> print(new_cube)
air_potential_temperature / (K)     (time: 3; model_level_number: 7)
    Dimension coordinates:
        time                            x                       -
        model_level_number              -                       x
    Auxiliary coordinates:
        forecast_period                 x                       -
        level_height                    -                       x
        sigma                           -                       x
    Derived coordinates:
        altitude                        -                       x
    Scalar coordinates:
        forecast_reference_time     2009-11-19 04:00:00
        grid_latitude               1.5145501 degrees, bound=(0.14430022, 2.8848) degrees
        grid_longitude              358.74948 degrees, bound=(357.494, 360.00497) degrees
        surface_altitude            399.625 m, bound=(-14.0, 813.25) m
    Cell methods:
        mean                        grid_longitude, grid_latitude
    Attributes:
        STASH                       m01s00i004
        source                      Data from Met Office Unified Model
        um_version                  7.3
```

Several examples of area averaging exist in the gallery which may be of interest, including an example on taking a *global area-weighted mean*.

13.2 Partially Reducing Data Dimensions

Instead of completely collapsing a dimension, other methods can be applied to reduce or filter the number of data points of a particular dimension.

13.2.1 Aggregation of Grouped Data

The *Cube.aggregated_by* operation combines data for all points with the same value of a given coordinate. To do this, you need a coordinate whose points take on only a limited set of different values – the *number* of these then determines the size of the reduced dimension. The *iris.coord_categorisation* module can be used to make such 'categorical' coordinates out of ordinary ones: The most common use is to aggregate data over regular *time intervals*, such as by calendar month or day of the week.

For example, let's create two new coordinates on the cube to represent the climatological seasons and the season year respectively:

```
import iris
import iris.coord_categorisation

filename = iris.sample_data_path('ostia_monthly.nc')
cube = iris.load_cube(filename, 'surface_temperature')
```

(continues on next page)

(continued from previous page)

```
iris.coord_categorisation.add_season(cube, 'time', name='clim_season')
iris.coord_categorisation.add_season_year(cube, 'time', name='season_year')
```

Note: The 'season year' is not the same as year number, because (e.g.) the months Dec11, Jan12 + Feb12 all belong to 'DJF-12'. See *iris.coord_categorisation.add_season_year()*.

Printing this cube now shows that two extra coordinates exist on the cube:

```
>>> print(cube)
surface_temperature / (K)           (time: 54; latitude: 18; longitude: 432)
    Dimension coordinates:
        time                            x             -             -
        latitude                        -             x             -
        longitude                       -             -             x
    Auxiliary coordinates:
        clim_season                     x             -             -
        forecast_reference_time         x             -             -
        season_year                     x             -             -
    Scalar coordinates:
        forecast_period             0 hours
    Cell methods:
        mean                        month, year
    Attributes:
        Conventions                 CF-1.5
        STASH                       m01s00i024
```

These two coordinates can now be used to aggregate by season and climate-year:

```
>>> annual_seasonal_mean = cube.aggregated_by(
...     ['clim_season', 'season_year'],
...     iris.analysis.MEAN)
>>> print(repr(annual_seasonal_mean))
<iris 'Cube' of surface_temperature / (K) (time: 19; latitude: 18; longitude: 432)>
```

The primary change in the cube is that the cube's data has been reduced in the 'time' dimension by aggregation (taking means, in this case). This has collected together all data points with the same values of season and season-year. The results are now indexed by the 19 different possible values of season and season-year in a new, reduced 'time' dimension.

We can see this by printing the first 10 values of season+year from the original cube: These points are individual months, so adjacent ones are often in the same season:

```
>>> for season, year in zip(cube.coord('clim_season')[:10].points,
...                         cube.coord('season_year')[:10].points):
...     print(season + ' ' + str(year))
mam 2006
mam 2006
jja 2006
jja 2006
jja 2006
son 2006
son 2006
son 2006
```

(continues on next page)

(continued from previous page)

```
djf 2007
djf 2007
```

Compare this with the first 10 values of the new cube's coordinates: All the points now have distinct season+year values:

```
>>> for season, year in zip(
...         annual_seasonal_mean.coord('clim_season')[:10].points,
...         annual_seasonal_mean.coord('season_year')[:10].points):
...     print(season + ' ' + str(year))
mam 2006
jja 2006
son 2006
djf 2007
mam 2007
jja 2007
son 2007
djf 2008
mam 2008
jja 2008
```

Because the original data started in April 2006 we have some incomplete seasons (e.g. there were only two months worth of data for 'mam-2006'). In this case we can fix this by removing all of the resultant 'times' which do not cover a three month period (note: judged here as > 3*28 days):

```
>>> tdelta_3mth = datetime.timedelta(hours=3*28*24.0)
>>> spans_three_months = lambda t: (t.bound[1] - t.bound[0]) > tdelta_3mth
>>> three_months_bound = iris.Constraint(time=spans_three_months)
>>> full_season_means = annual_seasonal_mean.extract(three_months_bound)
>>> full_season_means
<iris 'Cube' of surface_temperature / (K) (time: 17; latitude: 18; longitude: 432)>
```

The final result now represents the seasonal mean temperature for 17 seasons from jja-2006 to jja-2010:

```
>>> for season, year in zip(full_season_means.coord('clim_season').points,
...                         full_season_means.coord('season_year').points):
...     print(season + ' ' + str(year))
jja 2006
son 2006
djf 2007
mam 2007
jja 2007
son 2007
djf 2008
mam 2008
jja 2008
son 2008
djf 2009
mam 2009
jja 2009
son 2009
djf 2010
```

(continues on next page)

(continued from previous page)

```
mam 2010
jja 2010
```

CHAPTER

FOURTEEN

CUBE MATHS

The section *Navigating a Cube* highlighted that every cube has a data attribute; this attribute can then be manipulated directly:

```
cube.data -= 273.15
```

The problem with manipulating the data directly is that other metadata may become inconsistent; in this case the units of the cube are no longer what was intended. This example could be rectified by changing the units attribute:

```
cube.units = 'celsius'
```

Note: *iris.cube.Cube.convert_units()* can be used to automatically convert a cube's data and update its units attribute. So, the two steps above can be achieved by:

```
cube.convert_units('celsius')
```

In order to reduce the amount of metadata which becomes inconsistent, fundamental arithmetic operations such as addition, subtraction, division and multiplication can be applied directly to any cube.

14.1 Calculating the Difference Between Two Cubes

Let's load some air temperature which runs from 1860 to 2100:

```
filename = iris.sample_data_path('E1_north_america.nc')
air_temp = iris.load_cube(filename, 'air_temperature')
```

We can now get the first and last time slices using indexing (see *Subsetting a Cube* for a reminder):

```
t_first = air_temp[0, :, :]
t_last = air_temp[-1, :, :]
```

And finally we can subtract the two. The result is a cube of the same size as the original two time slices, but with the data representing their difference:

```
>>> print(t_last - t_first)
unknown / (K)                       (latitude: 37; longitude: 49)
    Dimension coordinates:
        latitude                            x              -
        longitude                           -              x
```

(continues on next page)

(continued from previous page)

```
    Scalar coordinates:
        forecast_reference_time     1859-09-01 06:00:00
        height                      1.5 m
    Attributes:
        Conventions                 CF-1.5
        Model scenario              E1
        source                      Data from Met Office Unified Model 6.05
```

Note: Notice that the coordinates "time" and "forecast_period" have been removed from the resultant cube; this is because these coordinates differed between the two input cubes.

14.2 Calculating a Cube Anomaly

In section *Cube Statistics* we discussed how the dimensionality of a cube can be reduced using the *Cube.collapsed* method to calculate a statistic over a dimension.

Let's use that method to calculate a mean of our air temperature time-series, which we'll then use to calculate a time mean anomaly and highlight the powerful benefits of cube broadcasting.

First, let's remind ourselves of the shape of our air temperature time-series cube:

```
>>> print(air_temp.summary(True))
air_temperature / (K)               (time: 240; latitude: 37; longitude: 49)
```

Now, we'll calculate the time-series mean using the *Cube.collapsed* method:

```
>>> air_temp_mean = air_temp.collapsed('time', iris.analysis.MEAN)
>>> print(air_temp_mean.summary(True))
air_temperature / (K)               (latitude: 37; longitude: 49)
```

As expected the *time* dimension has been collapsed, reducing the dimensionality of the resultant *air_temp_mean* cube. This time-series mean can now be used to calculate the time mean anomaly against the original time-series:

```
>>> anomaly = air_temp - air_temp_mean
>>> print(anomaly.summary(True))
unknown / (K)                       (time: 240; latitude: 37; longitude: 49)
```

Notice that the calculation of the *anomaly* involves subtracting a *2d* cube from a *3d* cube to yield a *3d* result. This is only possible because cube broadcasting is performed during cube arithmetic operations.

Cube broadcasting follows similar broadcasting rules as NumPy, but the additional richness of Iris coordinate meta-data provides an enhanced capability beyond the basic broadcasting behaviour of NumPy.

As the coordinate meta-data of a cube uniquely describes each dimension, it is possible to leverage this knowledge to identify the similar dimensions involved in a cube arithmetic operation. This essentially means that we are no longer restricted to performing arithmetic on cubes with identical shapes.

This extended broadcasting behaviour is highlighted in the following examples. The first of these shows that it is possible to involve the transpose of the air temperature time-series in an arithmetic operation with itself.

Let's first create the transpose of the air temperature time-series:

```
>>> air_temp_T = air_temp.copy()
>>> air_temp_T.transpose()
>>> print(air_temp_T.summary(True))
air_temperature / (K)               (longitude: 49; latitude: 37; time: 240)
```

Now add the transpose to the original time-series:

```
>>> result = air_temp + air_temp_T
>>> print(result.summary(True))
unknown / (K)                       (time: 240; latitude: 37; longitude: 49)
```

Notice that the *result* is the same dimensionality and shape as *air_temp*. Let's check that the arithmetic operation has calculated a result that we would intuitively expect:

```
>>> result == 2 * air_temp
True
```

Let's extend this example slightly, by taking a slice from the middle *latitude* dimension of the transpose cube:

```
>>> air_temp_T_slice = air_temp_T[:, 0, :]
>>> print(air_temp_T_slice.summary(True))
air_temperature / (K)               (longitude: 49; time: 240)
```

Compared to our original time-series, the *air_temp_T_slice* cube has one less dimension *and* its shape is different. However, this doesn't prevent us from performing cube arithmetic with it, thanks to the extended cube broadcasting behaviour:

```
>>> result = air_temp - air_temp_T_slice
>>> print(result.summary(True))
unknown / (K)                       (time: 240; latitude: 37; longitude: 49)
```

14.3 Combining Multiple Phenomena to Form a New One

Combining cubes of potential-temperature and pressure we can calculate the associated temperature using the equation:

$$T = \theta(\frac{p}{p_0})^{(287.05/1005)}$$

Where p is pressure, θ is potential temperature, p_0 is the potential temperature reference pressure and T is temperature.

First, let's load pressure and potential temperature cubes:

```
filename = iris.sample_data_path('colpex.pp')
phenomenon_names = ['air_potential_temperature', 'air_pressure']
pot_temperature, pressure = iris.load_cubes(filename, phenomenon_names)
```

In order to calculate $\frac{p}{p_0}$ we can define a coordinate which represents the standard reference pressure of 1000 hPa:

```
import iris.coords
p0 = iris.coords.AuxCoord(1000.0,
                          long_name='reference_pressure',
                          units='hPa')
```

We must ensure that the units of `pressure` and `p0` are the same, so convert the newly created coordinate using the *iris.coords.Coord.convert_units()* method:

```
p0.convert_units(pressure.units)
```

Now we can combine all of this information to calculate the air temperature using the equation above:

```
temperature = pot_temperature * ( (pressure / p0) ** (287.05 / 1005) )
```

Finally, the cube we have created needs to be given a suitable name:

```
temperature.rename('air_temperature')
```

The result could now be plotted using the guidance provided in the *Plotting a Cube* section.

A very similar example to this can be found in the examples section, with the title "Deriving Exner Pressure and Air Temperature".

14.4 Combining Units

It should be noted that when combining cubes by multiplication, division or power operations, the resulting cube will have a unit which is an appropriate combination of the constituent units. In the above example, since `pressure` and `p0` have the same unit, then `pressure / p0` has a dimensionless unit of `'1'`. Since `(pressure / p0)` has a unit of `'1'`, this does not change under power operations and so `( (pressure / p0) ** (287.05 / 1005) )` also has unit 1. Multiplying by a cube with unit `'1'` will preserve units, so the cube `temperature` will be given the same units as are in `pot_temperature`. It should be noted that some combinations of units, particularly those involving power operations, will not result in a valid unit and will cause the calculation to fail. For example, if a cube `a` had units `'m'` then `a ** 0.5` would result in an error since the square root of a meter has no meaningful unit (if `a` had units `'m2'` then `a ** 0.5` would result in a cube with units `'m'`).

Iris inherits units from cf_units which in turn inherits from UDUNITS. As well as the units UDUNITS provides, cf units also provides the units `'no-unit'` and `'unknown'`. A unit of `'no-unit'` means that the associated data is not suitable for describing with a unit, cf units considers `'no-unit'` unsuitable for combining and therefore any arithmetic done on a cube with `'no-unit'` will fail. A unit of `'unknown'` means that the unit describing the associated data cannot be determined. cf units and Iris will allow arithmetic on cubes with a unit of `'unknown'`, but the resulting cube will always have a unit of `'unknown'`. If a calculation is prevented because it would result in inappropriate units, it may be forced by setting the units of the original cubes to be `'unknown'`.

CHAPTER

FIFTEEN

CITING IRIS

If Iris played an important part in your research then please add us to your reference list by using one of the recommendations below.

15.1 BibTeX Entry

For example:

```
@manual{Iris,
author = {{Met Office}},
title = {Iris: A Python package for analysing and visualising meteorological and
↪oceanographic data sets},
edition = {v1.2},
year = {2010 - 2013},
address = {Exeter, Devon },
url = {http://scitools.org.uk/}
}
```

15.2 Downloaded Software

Suggested format:

```
ProductName. Version. ReleaseDate. Publisher. Location. DOIorURL. DownloadDate.
```

For example:

```
Iris. v1.2. 28-Feb-2013. Met Office. UK. https://github.com/SciTools/iris/archive/v1.2.0.
↪tar.gz 01-03-2013
```

15.3 Checked Out Software

Suggested format:

```
ProductName. Publisher. URL. CheckoutDate. RepositorySpecificCheckoutInformation.
```

For example:

```
Iris. Met Office. git@github.com:SciTools/iris.git 06-03-2013
```

Reference: [Jackson].

CHAPTER

SIXTEEN

CODE MAINTENANCE

From a user point of view "code maintenance" means ensuring that your existing working code stays working, in the face of changes to Iris.

16.1 Stability and Change

In practice, as Iris develops, most users will want to periodically upgrade their installed version to access new features or at least bug fixes.

This is obvious if you are still developing other code that uses Iris, or using code from other sources. However, even if you have only legacy code that remains untouched, some code maintenance effort is probably still necessary:

- On the one hand, *in principle*, working code will go on working, as long as you don't change anything else.
- However, such "version stasis" can easily become a growing burden, if you are simply waiting until an update becomes unavoidable, often that will eventually occur when you need to update some other software component, for some completely unconnected reason.

16.2 Principles of Change Management

When you upgrade software to a new version, you often find that you need to rewrite your legacy code, simply to keep it working.

In Iris, however, we aim to reduce code maintenance problems to an absolute minimum by following defined change management rules. These ensure that, *within a major release number* :

- you can be confident that your code will still work with subsequent minor releases
- you will be aware of future incompatibility problems in advance
- you can defer making code compatibility changes for some time, until it suits you

The above applies to minor version upgrades : e.g. code that works with version "1.4.2" should still work with a subsequent minor release such as "1.5.0" or "1.7.2".

A *major* release however, e.g. "v2.0.0" or "v3.0.0", can include more significant changes, including so-called "breaking" changes: This means that existing code may need to be modified to make it work with the new version.

Since breaking change can only occur at major releases, these are the *only* times we can alter or remove existing behaviours (even deprecated ones). This is what a major release is for : it enables the removal and replacement of old features.

Of course, even at a major release, we do still aim to keep breaking changes to a minimum.

CHAPTER

SEVENTEEN

INTRODUCTION

Some specific areas of Iris may require further explanation or a deep dive into additional detail above and beyond that offered by the *User Guide*.

This section provides a collection of additional material on focused topics that may be of interest to the more advanced or curious user.

Hint: If you wish further documentation on any specific topics or areas of Iris that are missing, then please let us know by raising a GitHub Documentation Issue on SciTools/Iris.

- *Metadata*
- *Lenient Metadata*
- *Lenient Cube Maths*

CHAPTER

EIGHTEEN

METADATA

This section provides a detailed overview of how your metadata is managed within Iris. In particular, it discusses what metadata is, where it fits into Iris, and more importantly how you can create, access, manipulate, and analyse your metadata.

All the finer details covered here may not be entirely relevant to your use case, but it's here if you ever need it. In fact, you may want to skip straight ahead to *Richer Metadata Behaviour*, and take it from there.

18.1 Introduction

As discussed in *Iris Data Structures*, Iris draws heavily from the NetCDF CF Metadata Conventions as a source for its data model, thus building on the widely recognised and understood terminology defined within those CF Conventions by the scientific community.

In *Iris Data Structures* we introduced several fundamental classes in Iris that care about your `data`, and also your `metadata` i.e., data about data. These are the *`Cube`*, the *`AuxCoord`*, and the *`DimCoord`*, all of which should be familiar to you now. In addition to these, Iris models several other classes of CF Conventions metadata. Namely,

- the *`AncillaryVariable`*, see Ancillary Data and Flags,
- the *`CellMeasure`*, see Cell Measures,
- the *`AuxCoordFactory`*, see Parametric Vertical Coordinate

Collectively, the aforementioned classes will be known here as the Iris CF Conventions classes.

Hint: If there are any CF Conventions metadata missing from Iris that you care about, then please let us know by raising a GitHub Issue on SciTools/iris

---

18.2 Common Metadata

Each of the Iris CF Conventions classes use **metadata** to define them and give them meaning.

The **metadata** used to define an Iris CF Conventions class is composed of individual **metadata members**, almost all of which reference specific CF Conventions terms. The individual metadata members used to define each of the Iris CF Conventions classes are shown in Table 18.1.

As Table 18.1 highlights, **specific** metadata is used to define and represent each Iris CF Conventions class. This means that metadata alone, can be used to easily **identify**, **compare** and **differentiate** between individual class instances.

For example, the collective metadata used to define an *AncillaryVariable* are the `standard_name`, `long_name`, `var_name`, `units`, and `attributes` members. Note that, these are the actual data attribute names of the metadata members on the Iris class.

Table 18.1: - Iris classes that model CF Conventions metadata

Metadata Members	*AncillaryVariable*	*AuxCoord*	*AuxCoordFactory*	*CellMeasure*	*Cube*	*DimCoord*	Metadata Members
`standard_name`	✓	✓	✓	✓	✓	✓	`standard_name`
`long_name`	✓	✓	✓	✓	✓	✓	`long_name`
`var_name`	✓	✓	✓	✓	✓	✓	`var_name`
`units`	✓	✓	✓	✓	✓	✓	`units`
`attributes`	✓	✓	✓	✓	✓	✓	`attributes`
`coord_system`		✓	✓			✓	`coord_system`
`climatological`		✓	✓			✓	`climatological`
`measure`				✓			`measure`
`cell_methods`					✓		`cell_methods`
`circular`						✓	`circular`

Note: The *var_name* and *circular* metadata members are Iris specific terms, rather than recognised CF Conventions terms.

18.3 Common Metadata API

As of Iris `3.0.0`, a unified treatment of metadata has been applied across each Iris class (Table 18.1) to allow users to easily manage and manipulate their metadata in a consistent way.

This is achieved through the `metadata` property, which allows you to manipulate the associated underlying metadata members as a collective. For example, given the following *Cube*,

```
>>> print(cube)
air_temperature / (K)               (time: 240; latitude: 37; longitude: 49)
    Dimension coordinates:
        time                            x              -              -
        latitude                        -              x              -
        longitude                       -              -              x
    Auxiliary coordinates:
        forecast_period                 x              -              -
    Scalar coordinates:
        forecast_reference_time     1859-09-01 06:00:00
        height                      1.5 m
    Cell methods:
        mean                        time (6 hour)
    Attributes:
        Conventions                 CF-1.5
        Model scenario              A1B
        STASH                       m01s03i236
        source                      Data from Met Office Unified Model 6.05
```

We can easily get all of the associated metadata of the *Cube* using the `metadata` property:

```
>>> cube.metadata
CubeMetadata(standard_name='air_temperature', long_name=None, var_name='air_temperature',
↪ units=Unit('K'), attributes={'Conventions': 'CF-1.5', 'STASH': STASH(model=1,␣
↪section=3, item=236), 'Model scenario': 'A1B', 'source': 'Data from Met Office Unified␣
↪Model 6.05'}, cell_methods=(CellMethod(method='mean', coord_names=('time',),␣
↪intervals=('6 hour',), comments=()),))
```

We can also inspect the `metadata` of the `longitude` *DimCoord* attached to the *Cube* in the same way:

```
>>> cube.coord("longitude").metadata
DimCoordMetadata(standard_name='longitude', long_name=None, var_name='longitude',␣
↪units=Unit('degrees'), attributes={}, coord_system=GeogCS(6371229.0),␣
↪climatological=False, circular=False)
```

Or use the `metadata` property again, but this time on the `forecast_period` *AuxCoord* attached to the *Cube*:

```
>>> cube.coord("forecast_period").metadata
CoordMetadata(standard_name='forecast_period', long_name=None, var_name='forecast_period
↪', units=Unit('hours'), attributes={}, coord_system=None, climatological=False)
```

Note that, the `metadata` property is available on each of the Iris CF Conventions class containers referenced in Table 18.1, and thus provides a **common** and **consistent** approach to managing your metadata, which we'll now explore a little more fully.

18.3.1 Metadata Classes

The `metadata` property will return an appropriate namedtuple metadata class for each Iris CF Conventions class container. The metadata class returned by each container class is shown in Table 18.2 below,

Table 18.2: - Iris namedtuple metadata classes

Container Class	Metadata Class
AncillaryVariable	*AncillaryVariableMetadata*
AuxCoord	*CoordMetadata*
AuxCoordFactory	*CoordMetadata*
CellMeasure	*CellMeasureMetadata*
Cube	*CubeMetadata*
DimCoord	*DimCoordMetadata*

Akin to the behaviour of a namedtuple, the metadata classes in Table 18.2 create **tuple-like** instances i.e., they provide a **snapshot** of the associated metadata member **values**, which are **not settable**, but they **may be mutable** depending on the data-type of the member. For example, given the following `metadata` of a *DimCoord*,

```
>>> longitude = cube.coord("longitude")
>>> metadata = longitude.metadata
>>> metadata
DimCoordMetadata(standard_name='longitude', long_name=None, var_name='longitude',␣
↪units=Unit('degrees'), attributes={}, coord_system=GeogCS(6371229.0),␣
↪climatological=False, circular=False)
```

The `metadata` member value **is** the same as the container class member value,

```
>>> metadata.attributes is longitude.attributes
True
>>> metadata.circular is longitude.circular
True
```

Like a namedtuple, the `metadata` member is **not settable**,

```
>>> metadata.attributes = {"grinning face": ""}
Traceback (most recent call last):
AttributeError: can't set attribute
```

However, for a dict member, it **is mutable**,

```
>>> metadata.attributes
{}
>>> longitude.attributes["grinning face"] = ""
>>> metadata.attributes
{'grinning face': ''}
>>> metadata.attributes["grinning face"] = ""
>>> longitude.attributes
{'grinning face': ''}
```

But `metadata` members with simple values are **not** mutable,

```
>>> metadata.circular
False
>>> longitude.circular = True
>>> metadata.circular
False
```

And of course, they're also **not** settable,

```
>>> metadata.circular = True
Traceback (most recent call last):
AttributeError: can't set attribute
```

Note that, the `metadata` property re-creates a **new** instance per invocation, with a **snapshot** of the container class metadata values at that point in time,

```
>>> longitude.metadata
DimCoordMetadata(standard_name='longitude', long_name=None, var_name='longitude',␣
↪units=Unit('degrees'), attributes={'grinning face': ''}, coord_system=GeogCS(6371229.
↪0), climatological=False, circular=True)
```

Skip ahead to *metadata assignment* for a fuller discussion on options how to **set** and **get** metadata on the instance of an Iris CF Conventions container class (Table 18.2).

18.3.2 Metadata Class Behaviour

As mentioned previously, the metadata classes in Table 18.2 inherit the behaviour of a namedtuple, and so act and feel like a namedtuple, just as you might expect. For example, given the following `metadata`,

```
>>> metadata
DimCoordMetadata(standard_name='longitude', long_name=None, var_name='longitude',␣
↪units=Unit('degrees'), attributes={'grinning face': ''}, coord_system=GeogCS(6371229.
↪0), climatological=False, circular=False)
```

We can use the namedtuple._make method to create a **new** *DimCoordMetadata* instance from an existing sequence or iterable. The number and order of the values used in the iterable must match that of the associated namedtuple._fields, which is discussed later,

```
>>> values = (1, 2, 3, 4, 5, 6, 7, 8)
>>> metadata._make(values)
DimCoordMetadata(standard_name=1, long_name=2, var_name=3, units=4, attributes=5, coord_
↪system=6, climatological=7, circular=8)
```

Note that, namedtuple._make is a class method, and so it is possible to create a **new** instance directly from the metadata class itself,

```
>>> from iris.common import DimCoordMetadata
>>> DimCoordMetadata._make(values)
DimCoordMetadata(standard_name=1, long_name=2, var_name=3, units=4, attributes=5, coord_
↪system=6, climatological=7, circular=8)
```

It is also possible to easily convert `metadata` to an dict using the namedtuple._asdict method. This can be particularly handy when a standard Python built-in container is required to represent your `metadata`,

```
>>> metadata._asdict()
{'standard_name': 'longitude', 'long_name': None, 'var_name': 'longitude', 'units': Unit(
↪'degrees'), 'attributes': {'grinning face': ''}, 'coord_system': GeogCS(6371229.0),
↪'climatological': False, 'circular': False}
```

Using the namedtuple._replace method allows you to create a new metadata class instance, but replacing specified members with **new** associated values,

```
>>> metadata
DimCoordMetadata(standard_name='longitude', long_name=None, var_name='longitude',␣
↪units=Unit('degrees'), attributes={'grinning face': ''}, coord_system=GeogCS(6371229.
↪0), climatological=False, circular=False)
>>> metadata._replace(standard_name=None, units=None)
DimCoordMetadata(standard_name=None, long_name=None, var_name='longitude', units=None,␣
↪attributes={'grinning face': ''}, coord_system=GeogCS(6371229.0), climatological=False,
↪ circular=False)
```

Another very useful method from the namedtuple toolkit is namedtuple._fields. This method returns a tuple of strings listing the `metadata` members, in a fixed order. This allows you to easily iterate over the metadata class members, for what ever purpose you may require, e.g.,

```
>>> metadata._fields
('standard_name', 'long_name', 'var_name', 'units', 'attributes', 'coord_system',
↪'climatological', 'circular')
```

```
>>> tuple([getattr(metadata, member) for member in metadata._fields])
('longitude', None, 'longitude', Unit('degrees'), {'grinning face': ''}, GeogCS(6371229.
↪0), False, False)
```

```
>>> tuple([getattr(metadata, member) for member in metadata._fields if member.endswith(
↪"name")])
('longitude', None, 'longitude')
```

Note that, namedtuple._fields is also a class method, so you don't need an instance to determine the members of a metadata class, e.g.,

```
>>> from iris.common import CubeMetadata
>>> CubeMetadata._fields
('standard_name', 'long_name', 'var_name', 'units', 'attributes', 'cell_methods')
```

Aside from the benefit of metadata classes inheriting behaviour and state from namedtuple, further additional rich behaviour is also available, which we explore next.

18.3.3 Richer Metadata Behaviour

The metadata classes from Table 18.2 support additional behaviour above and beyond that of the standard Python namedtuple, which allows you to easily **compare**, **combine**, **convert** and understand the **difference** between your `metadata` instances.

Metadata Equality

The metadata classes support both **equality** (`__eq__`) and **inequality** (`__ne__`), but no other rich comparison operators are implemented. This is simply because there is no obvious ordering to any collective of metadata members, as defined in Table 18.1.

For example, given the following *DimCoord*,

```
>>> longitude.metadata
DimCoordMetadata(standard_name='longitude', long_name=None, var_name='longitude',␣
↪units=Unit('degrees'), attributes={}, coord_system=GeogCS(6371229.0),␣
↪climatological=False, circular=False)
```

We can compare `metadata` using the == operator, as you may naturally expect,

```
>>> longitude.metadata == longitude.metadata
True
```

Or alternatively, using the `equal` method instead,

```
>>> longitude.metadata.equal(longitude.metadata)
True
```

Note that, the == operator (`__eq__`) and the `equal` method are both functionally equivalent. However, the `equal` method also provides a means to enable **lenient** equality, as discussed in *Lenient Equality*.

Strict Equality

By default, metadata class equality will perform a **strict** comparison between each associated `metadata` member. If **any** `metadata` member has a different value, then the result of the operation will be `False`. For example,

```
>>> other = longitude.metadata._replace(standard_name=None)
>>> other
DimCoordMetadata(standard_name=None, long_name=None, var_name='longitude', units=Unit(
↪'degrees'), attributes={}, coord_system=GeogCS(6371229.0), climatological=False,␣
↪circular=False)
>>> longitude.metadata == other
False
```

```
>>> longitude.attributes = {"grinning face": ""}
>>> other = longitude.metadata._replace(attributes={"grinning face":  ""})
>>> other
DimCoordMetadata(standard_name='longitude', long_name=None, var_name='longitude',␣
↪units=Unit('degrees'), attributes={'grinning face': ''}, coord_system=GeogCS(6371229.
↪0), climatological=False, circular=False)
>>> longitude.metadata == other
False
```

One further point worth highlighting is it is possible for NumPy scalars and arrays to appear in the `attributes` dict of some Iris metadata class instances. Normally, this would cause issues. For example,

```
>>> simply = {"one": np.int(1), "two": np.array([1.0, 2.0])}
>>> simply
{'one': 1, 'two': array([1., 2.])}
>>> fruity = {"one": np.int(1), "two": np.array([1.0, 2.0])}
>>> fruity
{'one': 1, 'two': array([1., 2.])}
>>> simply == fruity
Traceback (most recent call last):
ValueError: The truth value of an array with more than one element is ambiguous. Use a.
↪any() or a.all()
```

However, metadata class equality is rich enough to handle this eventuality,

```
>>> metadata1 = cube.metadata._replace(attributes=simply)
>>> metadata2 = cube.metadata._replace(attributes=fruity)
>>> metadata1
CubeMetadata(standard_name='air_temperature', long_name=None, var_name='air_temperature',
↪ units=Unit('K'), attributes={'one': 1, 'two': array([1., 2.])}, cell_
↪methods=(CellMethod(method='mean', coord_names=('time',), intervals=('6 hour',),␣
↪comments=()),))
>>> metadata2
CubeMetadata(standard_name='air_temperature', long_name=None, var_name='air_temperature',
↪ units=Unit('K'), attributes={'one': 1, 'two': array([1., 2.])}, cell_
↪methods=(CellMethod(method='mean', coord_names=('time',), intervals=('6 hour',),␣
↪comments=()),))
```

```
>>> metadata1 == metadata2
True
```

```
>>> metadata1
CubeMetadata(standard_name='air_temperature', long_name=None, var_name='air_temperature',
↪ units=Unit('K'), attributes={'one': 1, 'two': array([1., 2.])}, cell_
↪methods=(CellMethod(method='mean', coord_names=('time',), intervals=('6 hour',),␣
↪comments=()),))
>>> metadata2 = cube.metadata._replace(attributes={"one": np.int(1), "two": np.
↪array([1000.0, 2000.0])})
>>> metadata2
CubeMetadata(standard_name='air_temperature', long_name=None, var_name='air_temperature',
↪ units=Unit('K'), attributes={'one': 1, 'two': array([1000., 2000.])}, cell_
↪methods=(CellMethod(method='mean', coord_names=('time',), intervals=('6 hour',),␣
↪comments=()),))
>>> metadata1 == metadata2
False
```

Comparing Like With Like

So far in our journey through metadata class equality, we have only considered cases where the operands are instances of the **same** type. It is possible to compare instances of **different** metadata classes, but the result will always be `False`,

```
>>> cube.metadata == longitude.metadata
False
```

The reason different metadata classes cannot be compared is simply because each metadata class contains **different** members, as shown in Table 18.1. However, there is an exception to the rule…

Exception to the Rule

In general, **different** metadata classes cannot be compared, however support is provided for comparing *CoordMetadata* and *DimCoordMetadata* metadata classes. For example, consider the following *DimCoordMetadata*,

```
>>> latitude = cube.coord("latitude")
>>> latitude.metadata
DimCoordMetadata(standard_name='latitude', long_name=None, var_name='latitude',␣
↪units=Unit('degrees'), attributes={}, coord_system=GeogCS(6371229.0),␣
↪climatological=False, circular=False)
```

Next we create a new *CoordMetadata* instance from the *DimCoordMetadata* instance,

```
>>> kwargs = latitude.metadata._asdict()
>>> del kwargs["circular"]
>>> metadata = CoordMetadata(**kwargs)
>>> metadata
CoordMetadata(standard_name='latitude', long_name=None, var_name='latitude', units=Unit(
↪'degrees'), attributes={}, coord_system=GeogCS(6371229.0), climatological=False)
```

Hint: Alternatively, use the `from_metadata` class method instead, see *Metadata Conversion*.

Comparing the instances confirms that equality is indeed supported between *DimCoordMetadata* and *CoordMetadata* classes,

```
>>> latitude.metadata == metadata
True
```

The reason for this behaviour is primarily historical. The `circular` member has **never** been used by the `__eq__` operator when comparing an *AuxCoord* and a *DimCoord*. Therefore for consistency, this behaviour is also extended to `__eq__` for the associated container metadata classes.

However, note that the `circular` member **is used** by the `__eq__` operator when comparing one *DimCoord* to another. This also applies when comparing *DimCoordMetadata*.

This exception to the rule for *equality* also applies to the *difference* and *combine* methods of metadata classes.

Metadata Difference

Being able to compare metadata is valuable, especially when we have the convenience of being able to do this easily with metadata classes. However, when the result of comparing two metadata instances is `False`, it begs the question, "**what's the difference?**"

Well, this is where we pull the `difference` method out of the metadata toolbox. First, let's create some `metadata` to compare,

```
>>> longitude = cube.coord("longitude")
>>> longitude.metadata
DimCoordMetadata(standard_name='longitude', long_name=None, var_name='longitude',␣
↪units=Unit('degrees'), attributes={'grinning face': ''}, coord_system=GeogCS(6371229.
↪0), climatological=False, circular=False)
```

Now, we replace some members of the *DimCoordMetadata* with different values,

```
>>> from cf_units import Unit
>>> metadata = longitude.metadata._replace(long_name="lon", var_name="lon", units=Unit(
↪"radians"))
>>> metadata
DimCoordMetadata(standard_name='longitude', long_name='lon', var_name='lon', units=Unit(
↪'radians'), attributes={'grinning face': ''}, coord_system=GeogCS(6371229.0),␣
↪climatological=False, circular=False)
```

First, confirm that the `metadata` is different,

```
>>> longitude.metadata != metadata
True
```

As expected, the `metadata` is different. Now, let's answer the question, "**what's the difference?**",

```
>>> longitude.metadata.difference(metadata)
DimCoordMetadata(standard_name=None, long_name=(None, 'lon'), var_name=('longitude', 'lon
↪'), units=(Unit('degrees'), Unit('radians')), attributes=None, coord_system=None,␣
↪climatological=None, circular=None)
```

The `difference` method returns a *DimCoordMetadata* instance, when there is **at least** one `metadata` member with a different value, where,

- `None` means that there was **no** difference for the member,

- a tuple contains the two different associated values for the member

Given our example, only the `long_name`, `var_name` and `units` members have different values, as expected. Note that, the `difference` method **is not** commutative. The order of the tuple member values is the same order of the metadata class instances being compared, e.g., changing the `difference` instance order is reflected in the result,

```
>>> metadata.difference(longitude.metadata)
DimCoordMetadata(standard_name=None, long_name=('lon', None), var_name=('lon', 'longitude
↪'), units=(Unit('radians'), Unit('degrees')), attributes=None, coord_system=None,␣
↪climatological=None, circular=None)
```

Also, when the `metadata` being compared **is identical**, then `None` is simply returned,

```
>>> metadata.difference(metadata) is None
True
```

It's worth highlighting that for the `attributes` dict member, only those keys with **different values** or **missing keys** will be returned by the `difference` method. For example, let's customise the `attributes` member of the following *DimCoordMetadata*,

```
>>> attributes = {"grinning face": "", "neutral face": ""}
>>> longitude.attributes = attributes
>>> longitude.metadata
DimCoordMetadata(standard_name='longitude', long_name=None, var_name='longitude',␣
↪units=Unit('degrees'), attributes={'grinning face': '', 'neutral face': ''}, coord_
↪system=GeogCS(6371229.0), climatological=False, circular=False)
```

Then create another *DimCoordMetadata* with a different `attributes` dict, namely,

- the `grinning face` key has the **same value**,
- the `neutral face` key has a **different value**,
- the `upside-down face` key is **new**

```
>>> attributes = {"grinning face": "", "neutral face": "", "upside-down face": ""}
>>> metadata = longitude.metadata._replace(attributes=attributes)
>>> metadata
DimCoordMetadata(standard_name='longitude', long_name=None, var_name='longitude',␣
↪units=Unit('degrees'), attributes={'grinning face': '', 'neutral face': '', 'upside-
↪down face': ''}, coord_system=GeogCS(6371229.0), climatological=False, circular=False)
```

Now, let's compare the two above instances and see what `attributes` member differences we get,

```
>>> longitude.metadata.difference(metadata)
DimCoordMetadata(standard_name=None, long_name=None, var_name=None, units=None,␣
↪attributes=({'neutral face': ''}, {'neutral face': '', 'upside-down face': ''}), coord_
↪system=None, climatological=None, circular=None)
```

Diffing Like With Like

As discussed in *Comparing Like With Like*, it only makes sense to determine the `difference` between **similar** metadata class instances. However, note that the *exception to the rule* still applies here i.e., support is provided between *CoordMetadata* and *DimCoordMetadata* metadata classes.

For example, given the following *AuxCoord* and *DimCoord*,

```
>>> forecast_period = cube.coord("forecast_period")
>>> latitude = cube.coord("latitude")
```

We can inspect their associated `metadata`,

```
>>> forecast_period.metadata
CoordMetadata(standard_name='forecast_period', long_name=None, var_name='forecast_period
↪', units=Unit('hours'), attributes={}, coord_system=None, climatological=False)
>>> latitude.metadata
DimCoordMetadata(standard_name='latitude', long_name=None, var_name='latitude',␣
↪units=Unit('degrees'), attributes={}, coord_system=GeogCS(6371229.0),␣
↪climatological=False, circular=False)
```

Before comparing them to determine the values of metadata members that are different,

```
>>> forecast_period.metadata.difference(latitude.metadata)
CoordMetadata(standard_name=('forecast_period', 'latitude'), long_name=None, var_name=(
↪'forecast_period', 'latitude'), units=(Unit('hours'), Unit('degrees')),␣
↪attributes=None, coord_system=(None, GeogCS(6371229.0)), climatological=None)
```

```
>>> latitude.metadata.difference(forecast_period.metadata)
DimCoordMetadata(standard_name=('latitude', 'forecast_period'), long_name=None, var_
↪name=('latitude', 'forecast_period'), units=(Unit('degrees'), Unit('hours')),␣
↪attributes=None, coord_system=(GeogCS(6371229.0), None), climatological=None,␣
↪circular=(False, None))
```

In general, however, comparing **different** metadata classes will result in a `TypeError` being raised,

```
>>> cube.metadata.difference(longitude.metadata)
Traceback (most recent call last):
TypeError: Cannot differ 'CubeMetadata' with <class 'iris.common.metadata.
↪DimCoordMetadata'>.
```

Metadata Combination

So far we've seen how to *compare metadata*, and also how to determine the *difference between metadata*. Now we take the next step, and explore how to combine metadata together using the `combine` metadata class method.

For example, consider the following *CubeMetadata*,

```
>>> cube.metadata
CubeMetadata(standard_name='air_temperature', long_name=None, var_name='air_temperature',
↪ units=Unit('K'), attributes={'Conventions': 'CF-1.5', 'STASH': STASH(model=1,␣
↪section=3, item=236), 'Model scenario': 'A1B', 'source': 'Data from Met Office Unified␣
↪Model 6.05'}, cell_methods=(CellMethod(method='mean', coord_names=('time',),␣
↪intervals=('6 hour',), comments=()),))
```

We can perform the **identity function** by comparing the metadata with itself,

```
>>> metadata = cube.metadata.combine(cube.metadata)
>>> cube.metadata == metadata
True
```

As you might expect, combining identical metadata returns metadata that is also identical.

The `combine` method will always return **a new** metadata class instance, where each metadata member is either `None` or populated with a **common value**. Let's clarify this, by combining our above *CubeMetadata* with another instance that's identical apart from its `standard_name` member, which is replaced with a **different value**,

```
>>> metadata = cube.metadata._replace(standard_name="air_pressure_at_sea_level")
>>> metadata != cube.metadata
True
>>> metadata.combine(cube.metadata)
CubeMetadata(standard_name=None, long_name=None, var_name='air_temperature', units=Unit(
↪'K'), attributes={'STASH': STASH(model=1, section=3, item=236), 'source': 'Data from␣
↪Met Office Unified Model 6.05', 'Model scenario': 'A1B', 'Conventions': 'CF-1.5'},␣
↪cell_methods=(CellMethod(method='mean', coord_names=('time',), intervals=('6 hour',),␣
↪comments=()),))
```

The `combine` method combines metadata by performing a **strict** comparison between each of the associated metadata member values,

- if the values are **different**, then the combined result is `None`
- otherwise, the combined result is the **common value**

Let's reinforce this behaviour, but this time by combining metadata where the `attributes` dict member is different, where,

- the `STASH` and `source` keys are **missing**,
- the `Model scenario` key has the **same value**,
- the `Conventions` key has a **different value**,
- the `grinning face` key is **new**

```
>>> attributes = {"Model scenario": "A1B", "Conventions": "CF-1.8", "grinning face": "" }
>>> metadata = cube.metadata._replace(attributes=attributes)
>>> metadata != cube.metadata
True
>>> metadata.combine(cube.metadata).attributes
{'Model scenario': 'A1B'}
```

The combined result for the `attributes` member only contains those **common keys** with **common values**.

Note that, the `combine` method is **commutative**,

```
>>> cube.metadata.combine(metadata) == metadata.combine(cube.metadata)
True
```

Although, this is only the case when combining instances of the **same** metadata class. This is explored in a little further detail next.

Combine Like With Like

Akin to the *equal* and *difference* methods, only instances of **similar** metadata classes can be combined, otherwise a `TypeError` is raised,

```
>>> cube.metadata.combine(longitude.metadata)
Traceback (most recent call last):
TypeError: Cannot combine 'CubeMetadata' with <class 'iris.common.metadata.
↪DimCoordMetadata'>.
```

Again, however, the *exception to the rule* also applies here i.e., support is provided between *CoordMetadata* and *DimCoordMetadata* metadata classes.

For example, we can `combine` the metadata of the following *AuxCoord* and *DimCoord*,

```
>>> forecast_period = cube.coord("forecast_period")
>>> longitude = cube.coord("longitude")
```

First, let's see their associated metadata,

```
>>> forecast_period.metadata
CoordMetadata(standard_name='forecast_period', long_name=None, var_name='forecast_period
↪', units=Unit('hours'), attributes={}, coord_system=None, climatological=False)
>>> longitude.metadata
DimCoordMetadata(standard_name='longitude', long_name=None, var_name='longitude',␣
↪units=Unit('degrees'), attributes={}, coord_system=GeogCS(6371229.0),␣
↪climatological=False, circular=False)
```

Before combining their metadata together,

```
>>> forecast_period.metadata.combine(longitude.metadata)
CoordMetadata(standard_name=None, long_name=None, var_name=None, units=None, attributes=
↪{}, coord_system=None, climatological=False)
>>> longitude.metadata.combine(forecast_period.metadata)
DimCoordMetadata(standard_name=None, long_name=None, var_name=None, units=None,␣
↪attributes={}, coord_system=None, climatological=False, circular=None)
```

However, note that commutativity in this case cannot be honoured, for obvious reasons.

Metadata Conversion

In general, the *equal*, *difference*, and *combine* methods only support operations on instances of the same metadata class (see *exception to the rule*).

However, metadata may be converted from one metadata class to another using the `from_metadata` class method. For example, given the following *CubeMetadata*,

```
>>> cube.metadata
CubeMetadata(standard_name='air_temperature', long_name=None, var_name='air_temperature',
↪ units=Unit('K'), attributes={'Conventions': 'CF-1.5', 'STASH': STASH(model=1,␣
↪section=3, item=236), 'Model scenario': 'A1B', 'source': 'Data from Met Office Unified␣
↪Model 6.05'}, cell_methods=(CellMethod(method='mean', coord_names=('time',),␣
↪intervals=('6 hour',), comments=()),))
```

We can easily convert it to a *DimCoordMetadata* instance using `from_metadata`,

```
>>> DimCoordMetadata.from_metadata(cube.metadata)
DimCoordMetadata(standard_name='air_temperature', long_name=None, var_name='air_
↪temperature', units=Unit('K'), attributes={'Conventions': 'CF-1.5', 'STASH':␣
↪STASH(model=1, section=3, item=236), 'Model scenario': 'A1B', 'source': 'Data from Met␣
↪Office Unified Model 6.05'}, coord_system=None, climatological=None, circular=None)
```

By examining Table 18.1, we can see that the *Cube* and *DimCoord* container classes share the following common metadata members,

- standard_name,
- long_name,
- var_name,
- units,
- attributes

As such, all of these metadata members of the resultant *DimCoordMetadata* instance are populated from the associated *CubeMetadata* instance members. However, a *CubeMetadata* class does not contain the following *DimCoordMetadata* members,

- coords_system,
- climatological,
- circular

Thus these particular metadata members are set to None in the resultant *DimCoordMetadata* instance.

Note that, the from_metadata method is also available on a metadata class instance,

```
>>> longitude.metadata.from_metadata(cube.metadata)
DimCoordMetadata(standard_name='air_temperature', long_name=None, var_name='air_
↪temperature', units=Unit('K'), attributes={'Conventions': 'CF-1.5', 'STASH':␣
↪STASH(model=1, section=3, item=236), 'Model scenario': 'A1B', 'source': 'Data from Met␣
↪Office Unified Model 6.05'}, coord_system=None, climatological=None, circular=None)
```

Metadata Assignment

The metadata property available on each Iris CF Conventions container class (Table 18.2) can not only be used **to get** the metadata of an instance, but also **to set** the metadata on an instance.

For example, given the following *DimCoordMetadata* of the longitude coordinate,

```
>>> longitude.metadata
DimCoordMetadata(standard_name='longitude', long_name=None, var_name='longitude',␣
↪units=Unit('degrees'), attributes={}, coord_system=GeogCS(6371229.0),␣
↪climatological=False, circular=False)
```

We can assign to it directly using the *DimCoordMetadata* of the latitude coordinate,

```
>>> latitude.metadata
DimCoordMetadata(standard_name='latitude', long_name=None, var_name='latitude',␣
↪units=Unit('degrees'), attributes={}, coord_system=GeogCS(6371229.0),␣
↪climatological=False, circular=False)
>>> longitude.metadata = latitude.metadata
```

(continues on next page)

(continued from previous page)

```
>>> longitude.metadata
DimCoordMetadata(standard_name='latitude', long_name=None, var_name='latitude',␣
↪units=Unit('degrees'), attributes={}, coord_system=GeogCS(6371229.0),␣
↪climatological=False, circular=False)
```

Assign by Iterable

It is also possible to assign to the `metadata` property of an Iris CF Conventions container with an iterable containing the **correct number** of associated member values, e.g.,

```
>>> values = [getattr(latitude, member) for member in latitude.metadata._fields]
>>> longitude.metadata = values
>>> longitude.metadata
DimCoordMetadata(standard_name='latitude', long_name=None, var_name='latitude',␣
↪units=Unit('degrees'), attributes={}, coord_system=GeogCS(6371229.0),␣
↪climatological=False, circular=False)
```

Assign by Namedtuple

A namedtuple may also be used to assign to the `metadata` property of an Iris CF Conventions container. For example, let's first create a custom namedtuple class,

```
>>> from collections import namedtuple
>>> Metadata = namedtuple("Metadata", ["standard_name", "long_name", "var_name", "units",
↪ "attributes", "coord_system", "climatological", "circular"])
```

Now create an instance of this custom namedtuple class, and populate it,

```
>>> metadata = Metadata(*values)
>>> metadata
Metadata(standard_name='latitude', long_name=None, var_name='latitude', units=Unit(
↪'degrees'), attributes={}, coord_system=GeogCS(6371229.0), climatological=False,␣
↪circular=False)
```

Now we can use the custom namedtuple instance to assign directly to the metadata of the `longitude` coordinate,

```
>>> longitude.metadata = metadata
>>> longitude.metadata
DimCoordMetadata(standard_name='latitude', long_name=None, var_name='latitude',␣
↪units=Unit('degrees'), attributes={}, coord_system=GeogCS(6371229.0),␣
↪climatological=False, circular=False)
```

Assign by Mapping

It is also possible to assign to the `metadata` property using a mapping, such as a dict,

```
>>> mapping = latitude.metadata._asdict()
>>> mapping
{'standard_name': 'latitude', 'long_name': None, 'var_name': 'latitude', 'units': Unit(
↪'degrees'), 'attributes': {}, 'coord_system': GeogCS(6371229.0), 'climatological':␣
↪False, 'circular': False}
>>> longitude.metadata = mapping
>>> longitude.metadata
DimCoordMetadata(standard_name='latitude', long_name=None, var_name='latitude',␣
↪units=Unit('degrees'), attributes={}, coord_system=GeogCS(6371229.0),␣
↪climatological=False, circular=False)
```

Support is also provided for assigning a **partial** mapping, for example,

```
>>> longitude.metadata
DimCoordMetadata(standard_name='longitude', long_name=None, var_name='longitude',␣
↪units=Unit('degrees'), attributes={}, coord_system=GeogCS(6371229.0),␣
↪climatological=False, circular=False)
>>> longitude.metadata = dict(var_name="lat", units="radians", circular=True)
>>> longitude.metadata
DimCoordMetadata(standard_name='longitude', long_name=None, var_name='lat', units=Unit(
↪'radians'), attributes={}, coord_system=GeogCS(6371229.0), climatological=False,␣
↪circular=True)
```

Indeed, it's also possible to assign to the `metadata` property with a **different** metadata class instance,

```
>>> longitude.metadata
DimCoordMetadata(standard_name='longitude', long_name=None, var_name='longitude',␣
↪units=Unit('degrees'), attributes={}, coord_system=GeogCS(6371229.0),␣
↪climatological=False, circular=False)
>>> longitude.metadata = cube.metadata
>>> longitude.metadata
DimCoordMetadata(standard_name='air_temperature', long_name=None, var_name='air_
↪temperature', units=Unit('K'), attributes={'Conventions': 'CF-1.5', 'STASH':␣
↪STASH(model=1, section=3, item=236), 'Model scenario': 'A1B', 'source': 'Data from Met␣
↪Office Unified Model 6.05'}, coord_system=GeogCS(6371229.0), climatological=False,␣
↪circular=False)
```

Note that, only **common** metadata members will be assigned new associated values. All other metadata members will be left unaltered.

CHAPTER

NINETEEN

LENIENT METADATA

This section discusses lenient metadata; what it is, what it means, and how you can perform **lenient** rather than **strict** operations with your metadata.

19.1 Introduction

As discussed in *Metadata*, a rich, common metadata API is available within Iris that supports metadata *equality*, *difference*, *combination*, and also *conversion*.

The common metadata API is implemented through the `metadata` property on each of the Iris CF Conventions class containers (Table 18.2), and provides a common gateway for users to easily manage and manipulate their metadata in a consistent and unified way.

This is primarily all thanks to the metadata classes (Table 18.2) that support the necessary state and behaviour required by the common metadata API. Namely, it is the `equal` (`__eq__`), `difference` and `combine` methods that provide this rich metadata behaviour, all of which are explored more fully in *Metadata*.

19.2 Strict Behaviour

The feature that is common between the `equal`, `difference` and `combine` metadata class methods, is that they all perform **strict** metadata member comparisons **by default**.

The **strict** behaviour implemented by these methods can be summarised as follows, where `X` and `Y` are any objects that are non-identical,

Table 19.1: - Strict equality

Left	Right	`equal`
`X`	`Y`	`False`
`Y`	`X`	`False`
`X`	`X`	`True`
`X`	`None`	`False`
`None`	`X`	`False`

Table 19.2: - Strict difference

Left	Right	difference
X	Y	(X, Y)
Y	X	(Y, X)
X	X	None
X	None	(X, None)
None	X	(None, X)

Table 19.3: - Strict combination

Left	Right	combine
X	Y	None
Y	X	None
X	X	X
X	None	None
None	X	None

This type of **strict** behaviour does offer obvious benefit and value. However, it can be unnecessarily restrictive. For example, consider the metadata of the following `latitude` coordinate,

```
>>> latitude.metadata
DimCoordMetadata(standard_name='latitude', long_name=None, var_name='latitude',␣
↪units=Unit('degrees'), attributes={}, coord_system=GeogCS(6371229.0),␣
↪climatological=False, circular=False)
```

Now, let's create a doctored version of this metadata with a different `var_name`,

```
>>> metadata = latitude.metadata._replace(var_name=None)
>>> metadata
DimCoordMetadata(standard_name='latitude', long_name=None, var_name=None, units=Unit(
↪'degrees'), attributes={}, coord_system=GeogCS(6371229.0), climatological=False,␣
↪circular=False)
```

Clearly, these metadata are different,

```
>>> metadata != latitude.metadata
True
>>> metadata.difference(latitude.metadata)
DimCoordMetadata(standard_name=None, long_name=None, var_name=(None, 'latitude'),␣
↪units=None, attributes=None, coord_system=None, climatological=None, circular=None)
```

And yet, they both have the same `name`, which some may find slightly confusing (see *name()* for clarification)

```
>>> metadata.name()
'latitude'
>>> latitude.name()
'latitude'
```

Resolving this metadata inequality can only be overcome by ensuring that each metadata member precisely matches.

If your workflow demands such metadata rigour, then the default strict behaviour of the common metadata API will satisfy your needs. Typically though, such strictness is not necessary, and as of Iris `3.0.0` an alternative more practical behaviour is available.

19.3 Lenient Behaviour

Lenient metadata aims to offer a practical, common sense alternative to the strict rigour of the default Iris metadata behaviour. It is intended to be complementary, and suitable for those users with a more relaxed requirement regarding their metadata.

The lenient behaviour that is implemented as an alternative to the *strict equality*, *strict difference*, and *strict combination* can be summarised as follows,

Table 19.4: - Lenient equality

Left	Right	`equal`
`X`	`Y`	`False`
`Y`	`X`	`False`
`X`	`X`	`True`
`X`	`None`	`True`
`None`	`X`	`True`

Table 19.5: - Lenient difference

Left	Right	`difference`
`X`	`Y`	(`X`, `Y`)
`Y`	`X`	(`Y`, `X`)
`X`	`X`	`None`
`X`	`None`	`None`
`None`	`X`	`None`

Table 19.6: - Lenient combination

Left	Right	`combine`
`X`	`Y`	`None`
`Y`	`X`	`None`
`X`	`X`	`X`
`X`	`None`	`X`
`None`	`X`	`X`

Lenient behaviour is enabled for the `equal`, `difference`, and `combine` metadata class methods via the `lenient` keyword argument, which is `False` by default. Let's first explore some examples of lenient equality, difference and combination, before going on to clarify which metadata members adopt lenient behaviour for each of the metadata classes.

19.3.1 Lenient Equality

Lenient equality is enabled using the `lenient` keyword argument, therefore we are forced to use the `equal` method rather than the == operator (`__eq__`). Otherwise, the `equal` method and == operator are both functionally equivalent.

For example, consider the *previous strict example*, where two separate `latitude` coordinates are compared, each with different `var_name` members,

```
>>> metadata.equal(latitude.metadata, lenient=True)
True
```

Unlike strict comparison, lenient comparison is a little more forgiving. In this case, leniently comparing **something** with **nothing** (`None`) will always be `True`; it's the graceful compromise to the strict alternative.

So let's take the opportunity to reinforce this a little further before moving on, by leniently comparing different `attributes` dictionaries; a constant source of strict contention.

Firstly, populate the metadata of our `latitude` coordinate appropriately,

```
>>> attributes = {"grinning face": "", "neutral face": ""}
>>> latitude.attributes = attributes
>>> latitude.metadata
DimCoordMetadata(standard_name='latitude', long_name=None, var_name='latitude',␣
↪units=Unit('degrees'), attributes={'grinning face': '', 'neutral face': ''}, coord_
↪system=GeogCS(6371229.0), climatological=False, circular=False)
```

Then create another *DimCoordMetadata* with a different `attributes` dict, namely,

- the `grinning face` key is **missing**,
- the `neutral face` key has the **same value**, and
- the `upside-down face` key is **new**

```
>>> attributes = {"neutral face": "", "upside-down face": ""}
>>> metadata = latitude.metadata._replace(attributes=attributes)
>>> metadata
DimCoordMetadata(standard_name='latitude', long_name=None, var_name='latitude',␣
↪units=Unit('degrees'), attributes={'neutral face': '', 'upside-down face': ''}, coord_
↪system=GeogCS(6371229.0), climatological=False, circular=False)
```

Now, compare our metadata,

```
>>> metadata.equal(latitude.metadata)
False
>>> metadata.equal(latitude.metadata, lenient=True)
True
```

Again, lenient equality (Table 19.4) offers a more forgiving and practical alternative to strict behaviour.

19.3.2 Lenient Difference

Similar to *Lenient Equality*, the lenient `difference` method (Table 19.5) considers there to be no difference between comparing **something** with **nothing** (`None`). This working assumption is not naively applied to all metadata members, but rather a more pragmatic approach is adopted, as discussed later in *Lenient Members*.

Again, lenient behaviour for the `difference` metadata class method is enabled by the `lenient` keyword argument. For example, consider again the *previous strict example* involving our `latitude` coordinate,

```
>>> metadata.difference(latitude.metadata)
DimCoordMetadata(standard_name=None, long_name=None, var_name=(None, 'latitude'),␣
↪units=None, attributes=None, coord_system=None, climatological=None, circular=None)
>>> metadata.difference(latitude.metadata, lenient=True) is None
True
```

And revisiting our slightly altered `attributes` member comparison example, brings home the benefits of the lenient difference behaviour. So, given our `latitude` coordinate with its populated `attributes` dictionary,

```
>>> latitude.attributes
{'grinning face': '', 'neutral face': ''}
```

We create another *DimCoordMetadata* with a dissimilar `attributes` member, namely,

- the `grinning face` key is **missing**,
- the `neutral face` key has a **different value**, and
- the `upside-down face` key is **new**

```
>>> attributes = {"neutral face": "", "upside-down face": ""}
>>> metadata = latitude.metadata._replace(attributes=attributes)
>>> metadata
DimCoordMetadata(standard_name='latitude', long_name=None, var_name='latitude',␣
↪units=Unit('degrees'), attributes={'neutral face': '', 'upside-down face': ''}, coord_
↪system=GeogCS(6371229.0), climatological=False, circular=False)
```

Now comparing the strict and lenient behaviour for the `difference` method, highlights the change in how such dissimilar metadata is treated gracefully,

```
>>> metadata.difference(latitude.metadata).attributes
{'upside-down face': '', 'neutral face': ''}, {'neutral face': '', 'grinning face': ''}
>>> metadata.difference(latitude.metadata, lenient=True).attributes
{'neutral face': ''}, {'neutral face': ''}
```

19.3.3 Lenient Combination

The behaviour of the lenient `combine` metadata class method is outlined in Table 19.6, and as with *Lenient Equality* and *Lenient Difference* is enabled through the `lenient` keyword argument.

The difference in behaviour between **lenient** and *strict combination* is centred around the lenient handling of combining **something** with **nothing** (`None`) to return **something**. Whereas strict combination will only return a result from combining identical objects.

Again, this is best demonstrated through a simple example of attempting to combine partially overlapping `attributes` member dictionaries. For example, given the following `attributes` dictionary of our favoured `latitude` coordinate,

```
>>> latitude.attributes
{'grinning face': '', 'neutral face': ''}
```

We create another *DimCoordMetadata* with overlapping keys and values, namely,

- the `grinning face` key is **missing**,
- the `neutral face` key has the **same value**, and
- the `upside-down face` key is **new**

```
>>> attributes = {"neutral face": "", "upside-down face": ""}
>>> metadata = latitude.metadata._replace(attributes=attributes)
>>> metadata
DimCoordMetadata(standard_name='latitude', long_name=None, var_name='latitude',␣
↪units=Unit('degrees'), attributes={'neutral face': '', 'upside-down face': ''}, coord_
↪system=GeogCS(6371229.0), climatological=False, circular=False)
```

Comparing the strict and lenient behaviour of `combine` side-by-side highlights the difference in behaviour, and the advantages of lenient combination for more inclusive, richer metadata,

```
>>> metadata.combine(latitude.metadata).attributes
{'neutral face': ''}
>>> metadata.combine(latitude.metadata, lenient=True).attributes
{'neutral face': '', 'upside-down face': '', 'grinning face': ''}
```

19.3.4 Lenient Members

Lenient Behaviour is not applied regardlessly across all metadata members participating in a lenient `equal`, `difference` or `combine` operation. Rather, a more pragmatic application is employed based on the CF Conventions definition of the member, and whether being lenient would result in erroneous behaviour or interpretation.

Table 19.7: - Lenient member participation

Metadata Class	Member	Behaviour
All metadata classes†	`standard_name`	`lenient`‡
All metadata classes†	`long_name`	`lenient`‡
All metadata classes†	`var_name`	`lenient`‡
All metadata classes†	`units`	`strict`
All metadata classes†	`attributes`	`lenient`
CellMeasureMetadata	`measure`	`strict`
CoordMetadata, *DimCoordMetadata*	`coord_system`	`strict`
CoordMetadata, *DimCoordMetadata*	`climatological`	`strict`
CubeMetadata	`cell_methods`	`strict`
DimCoordMetadata	`circular`	`strict` §

Key

† - Applies to all metadata classes including *AncillaryVariableMetadata*, which has no other specialised members

‡ - See *Special Lenient Name Behaviour* for `standard_name`, `long_name`, and `var_name`

§ - The `circular` is ignored for operations between *CoordMetadata* and *DimCoordMetadata*

In summary, only `standard_name`, `long_name`, `var_name` and the `attributes` members are treated leniently. All other members are considered to represent fundamental metadata that cannot, by their nature, be consider equivalent to metadata that is missing or `None`. For example, a *Cube* with `units` of `ms-1` cannot be considered equivalent to another *Cube* with `units` of `unknown`; this would be a false and dangerous scientific assumption to make.

Similar arguments can be made for the `measure`, `coord_system`, `climatological`, `cell_methods`, and `circular` members, all of which are treated with strict behaviour, regardlessly.

Special Lenient Name Behaviour

The `standard_name`, `long_name` and `var_name` have a closer association with each other compared to all other metadata members, as they all underpin the functionality provided by the *`name()`* method. It is imperative that the *`name()`* derived from metadata remains constant for strict and lenient equality alike.

As such, these metadata members have an additional layer of behaviour enforced during *Lenient Equality* in order to ensure that the identity or name of metadata does not change due to a side-effect of lenient comparison.

For example, if simple *lenient equality* behaviour was applied to the `standard_name`, `long_name` and `var_name`, the following would be considered **not** equal,

Member	Left	Right
`standard_name`	`None`	`latitude`
`long_name`	`latitude`	`None`
`var_name`	`lat`	`latitude`

Both the **Left** and **Right** metadata would have the same *`name()`* by definition i.e., `latitude`. However, lenient equality would fail due to the difference in `var_name`.

To account for this, lenient equality is performed by two simple consecutive steps:

- ensure that the result returned by the *`name()`* method is the same for the metadata being compared, then
- only perform *lenient equality* between the `standard_name` and `long_name` i.e., the `var_name` member is **not** compared explicitly, as its value may have been accounted for through *`name()`* equality

CHAPTER

TWENTY

LENIENT CUBE MATHS

This section provides an overview of lenient cube maths. In particular, it explains what lenient maths involves, clarifies how it differs from normal or strict cube maths, and demonstrates how you can exercise fine control over whether your cube maths operations are lenient or strict.

Note that, lenient cube maths is the default behaviour of Iris from version `3.0.0`.

20.1 Introduction

Lenient maths stands somewhat on the shoulders of giants. If you've not already done so, you may want to recap the material discussed in the following sections,

- *Cube Maths*,
- *Metadata*,
- *Lenient Metadata*

In addition to this, cube maths leans heavily on the *resolve* module, which provides the necessary infrastructure required by Iris to analyse and combine each *Cube* operand involved in a maths operation into the resultant *Cube*. It may be worth while investing some time to understand how the *Resolve* class underpins cube maths, and consider how it may be used in general to combine or resolve cubes together.

Given these prerequisites, recall that *lenient behaviour* introduced and discussed the concept of lenient metadata; a more pragmatic and forgiving approach to *comparing*, *combining* and understanding the *differences* between your metadata (Table 18.1). The lenient metadata philosophy introduced there is extended to cube maths, with the view to also preserving as much common coordinate (Table 18.2) information, as well as common metadata, between the participating *Cube* operands as possible.

Let's consolidate our understanding of lenient and strict cube maths through a practical worked example, which we'll explore together next.

20.2 Lenient Example

Consider the following *Cube* of `air_potential_temperature`, which has an atmosphere hybrid height parametric vertical coordinate, and represents the output of an low-resolution global atmospheric `experiment`,

```
>>> print(experiment)
air_potential_temperature / (K)             (model_level_number: 15; grid_latitude: 100;␣
↪grid_longitude: 100)
    Dimension coordinates:
        model_level_number                                  x                    -      ␣
↪                -
```

(continues on next page)

(continued from previous page)

```
        grid_latitude                                       -                   x      ␣
↪               -
        grid_longitude                                      -                   -      ␣
↪               x
    Auxiliary coordinates:
        atmosphere_hybrid_height_coordinate                 x                   -      ␣
↪               -
        sigma                                               x                   -      ␣
↪               -
        surface_altitude                                    -                   x      ␣
↪               x
    Derived coordinates:
        altitude                                            x                   x      ␣
↪               x
    Scalar coordinates:
        forecast_period                 0.0 hours
        forecast_reference_time         2009-09-09 17:10:00
        time                            2009-09-09 17:10:00
    Attributes:
        Conventions                     CF-1.5
        STASH                           m01s00i004
        experiment-id                   RT3 50
        source                          Data from Met Office Unified Model 7.04
```

Consider also the following *Cube*, which has the same global spatial extent, and acts as a `control`,

```
>>> print(control)
air_potential_temperature / (K)     (grid_latitude: 100; grid_longitude: 100)
    Dimension coordinates:
        grid_latitude                           x                   -
        grid_longitude                          -                   x
    Scalar coordinates:
        model_level_number          1
        time                        2009-09-09 17:10:00
    Attributes:
        Conventions                 CF-1.7
        STASH                       m01s00i004
        source                      Data from Met Office Unified Model 7.04
```

Now let's subtract these cubes in order to calculate a simple `difference`,

```
>>> difference = experiment - control
>>> print(difference)
unknown / (K)                           (model_level_number: 15; grid_latitude: 100;␣
↪grid_longitude: 100)
    Dimension coordinates:
        model_level_number                                  x                   -      ␣
↪               -
        grid_latitude                                       -                   x      ␣
↪               -
        grid_longitude                                      -                   -      ␣
↪               x
```

(continues on next page)

(continued from previous page)

```
    Auxiliary coordinates:
        atmosphere_hybrid_height_coordinate                 x                   -      ␣
↪               -
        sigma                                               x                   -      ␣
↪               -
        surface_altitude                                    -                   x      ␣
↪               x
    Derived coordinates:
        altitude                                            x                   x      ␣
↪               x
    Scalar coordinates:
        forecast_period                     0.0 hours
        forecast_reference_time             2009-09-09 17:10:00
        time                                2009-09-09 17:10:00
    Attributes:
        experiment-id                       RT3 50
        source                              Data from Met Office Unified Model 7.04
```

Note that, cube maths automatically takes care of broadcasting the dimensionality of the `control` up to that of the `experiment`, in order to calculate the `difference`. This is performed only after ensuring that both the **dimension coordinates** `grid_latitude` and `grid_longitude` are first *leniently equivalent*.

As expected, the resultant `difference` contains the *`HybridHeightFactory`* and all it's associated **auxiliary coordinates**. However, the **scalar coordinates** have been leniently combined to preserve as much coordinate information as possible, and the `attributes` dictionaries have also been leniently combined. In addition, see what further *rationalisation* is always performed by cube maths on the resultant metadata and coordinates.

Also, note that the `model_level_number` **scalar coordinate** from the `control` has be superseded by the similarly named **dimension coordinate** from the `experiment` in the resultant `difference`.

Now let's compare and contrast this lenient result with the strict alternative. But before we do so, let's first clarify how to control the behaviour of cube maths.

20.3 Control the Behaviour

As stated earlier, lenient cube maths is the default behaviour from Iris `3.0.0`. However, this behaviour may be controlled via the thread-safe `LENIENT["maths"]` runtime option,

```
>>> from iris.common import LENIENT
>>> print(LENIENT)
Lenient(maths=True)
```

Which may be set and applied globally thereafter for Iris within the current thread of execution,

```
>>> LENIENT["maths"] = False
>>> print(LENIENT)
Lenient(maths=False)
```

Or alternatively, temporarily alter the behaviour of cube maths only within the scope of the `LENIENT` context manager,

```
>>> print(LENIENT)
Lenient(maths=True)
```

(continues on next page)

(continued from previous page)

```
>>> with LENIENT.context(maths=False):
...     print(LENIENT)
...
Lenient(maths=False)
>>> print(LENIENT)
Lenient(maths=True)
```

20.4 Strict Example

Now that we know how to control the underlying behaviour of cube maths, let's return to our *lenient example*, but this time perform **strict** cube maths instead,

```
>>> with LENIENT.context(maths=False):
...     difference = experiment - control
...
>>> print(difference)
unknown / (K)                               (model_level_number: 15; grid_latitude: 100;␣
↪grid_longitude: 100)
    Dimension coordinates:
        model_level_number                                      x                   -      ␣
↪               -
        grid_latitude                                           -                   x      ␣
↪               -
        grid_longitude                                          -                   -      ␣
↪               x
    Auxiliary coordinates:
        atmosphere_hybrid_height_coordinate                     x                   -      ␣
↪               -
        sigma                                                   x                   -      ␣
↪               -
        surface_altitude                                        -                   x      ␣
↪               x
    Derived coordinates:
        altitude                                                x                   x      ␣
↪               x
    Scalar coordinates:
        time                                2009-09-09 17:10:00
    Attributes:
        source                              Data from Met Office Unified Model 7.04
```

Although the numerical result of this strict cube maths operation is identical, it is not as rich in metadata as the *lenient alternative*. In particular, it does not contain the `forecast_period` and `forecast_reference_time` **scalar coordinates**, or the `experiment-id` in the `attributes` dictionary.

This is because strict cube maths, in general, will only return common metadata and common coordinates that are *strictly equivalent*.

20.5 Finer Detail

In general, if you want to preserve as much metadata and coordinate information as possible during cube maths, then opt to use the default lenient behaviour. Otherwise, favour the strict alternative if you require to enforce precise metadata and coordinate commonality.

The following information may also help you decide whether lenient cube maths best suits your use case,

- lenient behaviour uses *lenient equality* to match the metadata of coordinates, which is more tolerant to certain metadata differences,
- lenient behaviour uses *lenient combination* to create the metadata of coordinates on the resultant *Cube*,
- lenient behaviour will attempt to cover each dimension with a *DimCoord* in the resultant *Cube*, even though only one *Cube* operand may describe that dimension,
- lenient behaviour will attempt to include **auxiliary coordinates** in the resultant *Cube* that exist on only one *Cube* operand,
- lenient behaviour will attempt to include **scalar coordinates** in the resultant *Cube* that exist on only one *Cube* operand,
- lenient behaviour will add a coordinate to the resultant *Cube* with **bounds**, even if only one of the associated matching coordinates from the *Cube* operands has **bounds**,
- strict and lenient behaviour both require that the **points** and **bounds** of matching coordinates from *Cube* operands must be strictly equivalent. However, mismatching **bounds** of **scalar coordinates** are ignored i.e., a scalar coordinate that is common to both *Cube* operands, with equivalent **points** but different **bounds**, will be added to the resultant *Cube* with but with **no bounds**

Additionally, cube maths will always perform the following rationalisation of the resultant *Cube*,

- clear the `standard_name`, `long_name` and `var_name`, defaulting the *name()* to `unknown`,
- clear the *cell_methods*,
- clear the *cell_measures()*,
- clear the *ancillary_variables()*,
- clear the `STASH` key from the *attributes* dictionary,
- assign the appropriate *units*

CHAPTER

TWENTYONE

GETTING INVOLVED

Iris is an Open Source project hosted on Github and as such anyone with a GitHub account may create an issue on our Iris GitHub project page for raising:

- bug reports
- feature requests
- documentation improvements

The Iris GitHub project has been configured to use templates for each of the above issue types when creating a new issue to ensure the appropriate information is provided.

A pull request may also be created by anyone who has become a **contributor** to Iris. Permissions to merge pull requests to the `main` branch are only given to **core developers** of Iris, this is to ensure a measure of control.

To get started we suggest reading recent issues and pull requests for Iris.

If you are new to using GitHub we recommend reading the GitHub getting started

---

Note: For more information on becoming a contributor including a link to the Contributors Licence Agreement (CLA) see the Governance section of the SciTools ogranization web site.

CHAPTER

TWENTYTWO

WORKING WITH IRIS SOURCE CODE

22.1 Introduction

These pages describe a git and github workflow for the Iris project.

This is not a comprehensive git reference, it's just a workflow for our own project. It's tailored to the github hosting service. You may well find better or quicker ways of getting stuff done with git, but these should get you started.

Tip: Please see the official git documentation for a complete list of git **commands** and **cheat sheets**.

22.2 Making Your own Copy (fork) of Iris

You need to do this only once. The instructions here are very similar to the instructions at http://help.github.com/forking/, please see that page for more detail. We're repeating some of it here just to give the specifics for the Iris project, and to suggest some default names.

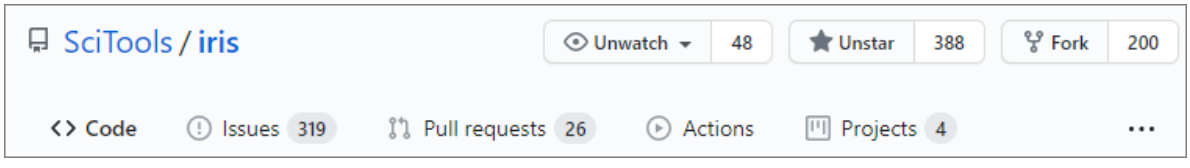
22.2.1 Set up and Configure a Github Account

If you don't have a github account, go to the github page, and make one.

You then need to configure your account to allow write access, see the generating sss keys for GitHub help on github help.

22.2.2 Create Your own Forked Copy of Iris

1. Log into your github account.
2. Go to the Iris github home at Iris github.
3. Click on the *fork* button:



Now, after a short pause, you should find yourself at the home page for your own forked copy of Iris.

22.3 Set up Your Fork

First you follow the instructions for *Making Your own Copy (fork) of Iris*.

22.3.1 Overview

```
git clone git@github.com:your-user-name/iris.git
cd iris
git remote add upstream git://github.com/SciTools/iris.git
```

22.3.2 In Detail

Clone Your Fork

1. Clone your fork to the local computer with `git clone git@github.com:your-user-name/iris.git`
2. Change directory to your new repo: `cd iris`. Then `git branch -a` to show you all branches. You'll get something like:

   ```
   * main
   remotes/origin/main
   ```

   This tells you that you are currently on the `main` branch, and that you also have a `remote` connection to `origin/main`. What remote repository is `remote/origin`? Try `git remote -v` to see the URLs for the remote. They will point to your github fork.

   Now you want to connect to the upstream Iris github repository, so you can merge in changes from trunk.

Linking Your Repository to the Upstream Repo

```
cd iris
git remote add upstream git://github.com/SciTools/iris.git
```

`upstream` here is just the arbitrary name we're using to refer to the main Iris repository at Iris github.

Note that we've used `git://` for the URL rather than `git@`. The `git://` URL is read only. This means we that we can't accidentally (or deliberately) write to the upstream repo, and we are only going to use it to merge into our own code.

Just for your own satisfaction, show yourself that you now have a new 'remote', with `git remote -v`, giving you something like:

```
upstream     git://github.com/SciTools/iris.git (fetch)
upstream     git://github.com/SciTools/iris.git (push)
origin       git@github.com:your-user-name/iris.git (fetch)
origin       git@github.com:your-user-name/iris.git (push)
```

22.4 Configure Git

22.4.1 Overview

Your personal git configurations are saved in the `.gitconfig` file in your home directory.

Here is an example `.gitconfig` file:

```
[user]
        name = Your Name
        email = you@yourdomain.example.com

[alias]
        ci = commit -a
        co = checkout
        st = status
        stat = status
        br = branch
        wdiff = diff --color-words

[core]
        editor = vim

[merge]
        summary = true
```

You can edit this file directly or you can use the `git config --global` command:

```
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
git config --global alias.ci "commit -a"
git config --global alias.co checkout
git config --global alias.st "status -a"
git config --global alias.stat "status -a"
git config --global alias.br branch
git config --global alias.wdiff "diff --color-words"
git config --global core.editor vim
git config --global merge.summary true
```

To set up on another computer, you can copy your `~/.gitconfig` file, or run the commands above.

22.4.2 In Detail

user.name and user.email

It is good practice to tell git who you are, for labelling any changes you make to the code. The simplest way to do this is from the command line:

```
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
```

This will write the settings into your git configuration file, which should now contain a user section with your name and email:

```
[user]
      name = Your Name
      email = you@yourdomain.example.com
```

Of course you'll need to replace `Your Name` and `you@yourdomain.example.com` with your actual name and email address.

Aliases

You might well benefit from some aliases to common commands.

For example, you might well want to be able to shorten `git checkout` to `git co`. Or you may want to alias `git diff --color-words` (which gives a nicely formatted output of the diff) to `git wdiff`

The following `git config --global` commands:

```
git config --global alias.ci "commit -a"
git config --global alias.co checkout
git config --global alias.st "status -a"
git config --global alias.stat "status -a"
git config --global alias.br branch
git config --global alias.wdiff "diff --color-words"
```

will create an `alias` section in your `.gitconfig` file with contents like this:

```
[alias]
        ci = commit -a
        co = checkout
        st = status -a
        stat = status -a
        br = branch
        wdiff = diff --color-words
```

Editor

You may also want to make sure that your editor of choice is used

```
git config --global core.editor vim
```

Merging

To enforce summaries when doing merges (`~/.gitconfig` file again):

```
[merge]
   log = true
```

Or from the command line:

```
git config --global merge.log true
```

Fancy Log Output

This is a very nice alias to get a fancy log output; it should go in the `alias` section of your `.gitconfig` file:

```
lg = log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr)
↪%C(bold blue)[%an]%Creset' --abbrev-commit --date=relative
```

You use the alias with:

```
git lg
```

and it gives graph / text output something like this (but with color!):

```
* 6d8e1ee - (HEAD, origin/my-fancy-feature, my-fancy-feature) NF - a fancy file (45␣
↪minutes ago) [Matthew Brett]
*   d304a73 - (origin/placeholder, placeholder) Merge pull request #48 from hhuuggoo/
↪main (2 weeks ago) [Jonathan Terhorst]
|\
| * 4aff2a8 - fixed bug 35, and added a test in test_bugfixes (2 weeks ago) [Hugo]
|/
* a7ff2e5 - Added notes on discussion/proposal made during Data Array Summit. (2 weeks␣
↪ago) [Corran Webster]
* 68f6752 - Initial implimentation of AxisIndexer - uses 'index_by' which needs to be␣
↪changed to a call on an Axes object - this is all very sketchy right now. (2 weeks␣
↪ago) [Corr
*   376adbd - Merge pull request #46 from terhorst/main (2 weeks ago) [Jonathan Terhorst]
|\
| * b605216 - updated joshu example to current api (3 weeks ago) [Jonathan Terhorst]
| * 2e991e8 - add testing for outer ufunc (3 weeks ago) [Jonathan Terhorst]
| * 7beda5a - prevent axis from throwing an exception if testing equality with non-axis␣
↪object (3 weeks ago) [Jonathan Terhorst]
| * 65af65e - convert unit testing code to assertions (3 weeks ago) [Jonathan Terhorst]
| *   956fbab - Merge remote-tracking branch 'upstream/main' (3 weeks ago) [Jonathan␣
↪Terhorst]
| |\
| |/
```

22.5 Development Workflow

You already have your own forked copy of the iris repository, by following *Making Your own Copy (fork) of Iris*. You have *Set up Your Fork*. You have configured git by following *Configure Git*. Now you are ready for some real work.

22.5.1 Workflow Summary

In what follows we'll refer to the upstream iris `main` branch, as "trunk".

- Don't use your `main` (that is on your fork) branch for anything. Consider deleting it.
- When you are starting a new set of changes, fetch any changes from trunk, and start a new *feature branch* from that.
- Make a new branch for each separable set of changes — "one task, one branch".
- Name your branch for the purpose of the changes - e.g. `bugfix-for-issue-14` or `refactor-database-code`.
- If you can possibly avoid it, avoid merging trunk or any other branches into your feature branch while you are working.
- If you do find yourself merging from trunk, consider *Rebasing on Trunk*
- Ask on the Iris GitHub Discussions if you get stuck.
- Ask for code review!

This way of working helps to keep work well organized, with readable history. This in turn makes it easier for project maintainers (that might be you) to see what you've done, and why you did it.

See linux git workflow for some explanation.

22.5.2 Consider Deleting Your Main Branch

It may sound strange, but deleting your own `main` branch can help reduce confusion about which branch you are on. See deleting master on github for details.

22.5.3 Update the Mirror of Trunk

First make sure you have done *Linking Your Repository to the Upstream Repo*.

From time to time you should fetch the upstream (trunk) changes from github:

```
git fetch upstream
```

This will pull down any commits you don't have, and set the remote branches to point to the right commit. For example, 'trunk' is the branch referred to by (remote/branchname) `upstream/main` - and if there have been commits since you last checked, `upstream/main` will change after you do the fetch.

22.5.4 Make a New Feature Branch

When you are ready to make some changes to the code, you should start a new branch. Branches that are for a collection of related edits are often called 'feature branches'.

Making an new branch for each set of related changes will make it easier for someone reviewing your branch to see what you are doing.

Choose an informative name for the branch to remind yourself and the rest of us what the changes in the branch are for. For example `add-ability-to-fly`, or `buxfix-for-issue-42`.

```
# Update the mirror of trunk
git fetch upstream
# Make new feature branch starting at current trunk
git branch my-new-feature upstream/main
git checkout my-new-feature
```

Generally, you will want to keep your feature branches on your public github fork of iris. To do this, you git push this new branch up to your github repo. Generally (if you followed the instructions in these pages, and by default), git will have a link to your github repo, called `origin`. You push up to your own repo on github with:

```
git push origin my-new-feature
```

In git >= 1.7 you can ensure that the link is correctly set by using the `--set-upstream` option:

```
git push --set-upstream origin my-new-feature
```

From now on git will know that `my-new-feature` is related to the `my-new-feature` branch in the github repo.

22.5.5 The Editing Workflow

Overview

```
# hack hack
git add my_new_file
git commit -am 'NF - some message'
git push
```

In More Detail

1. Make some changes
2. See which files have changed with `git status` (see git status). You'll see a listing like this one:

```
# On branch ny-new-feature
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#  modified:   README
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#  INSTALL
no changes added to commit (use "git add" and/or "git commit -a")
```

3. Check what the actual changes are with `git diff` (git diff).
4. Add any new files to version control `git add new_file_name` (see git add).
5. To commit all modified files into the local copy of your repo, do `git commit -am 'A commit message'`. Note the `-am` options to `commit`. The `m` flag just signals that you're going to type a message on the command line. The `a` flag will automatically stage all files that have been modified and deleted.

6. To push the changes up to your forked repo on github, do a `git push` (see git push).

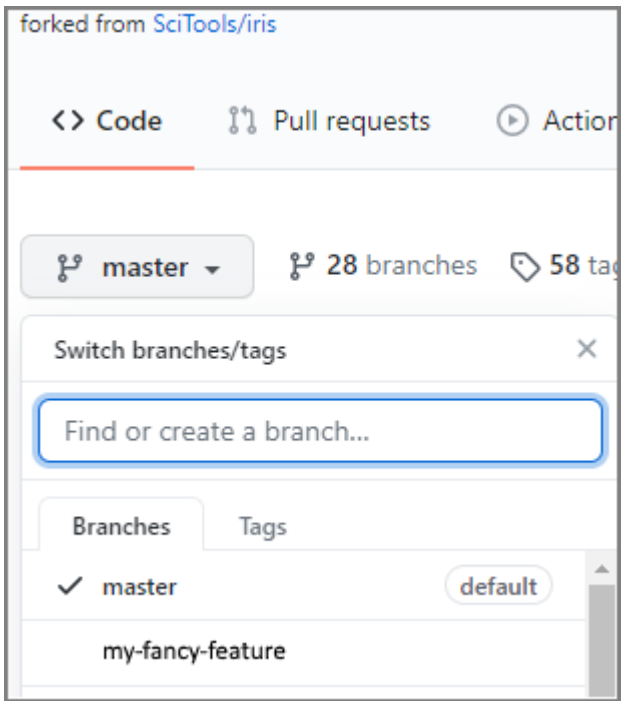
22.5.6 Testing Your Changes

Once you are happy with your changes, work thorough the *Pull Request Checklist* and make sure your branch passes all the relevant tests.

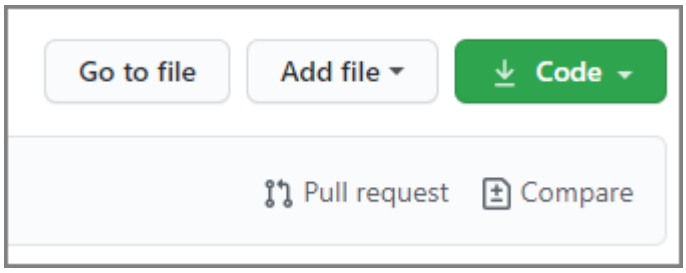
22.5.7 Ask for Your Changes to be Reviewed or Merged

When you are ready to ask for someone to review your code and consider a merge:

1. Go to the URL of your forked repo, say `http://github.com/your-user-name/iris`.
2. Use the 'Switch Branches' dropdown menu near the top left of the page to select the branch with your changes:



3. Click on the 'Pull request' button:



Enter a title for the set of changes, and some explanation of what you've done. Say if there is anything you'd like particular attention for - like a complicated change or some code you are not happy with.

If you don't think your request is ready to be merged, just say so in your pull request message. This is still a good way of getting some preliminary code review.

22.5.8 Some Other Things you Might Want to do

Delete a Branch on Github

```
git checkout main
# delete branch locally
git branch -D my-unwanted-branch
# delete branch on github
git push origin :my-unwanted-branch
```

Note the colon `:` before `test-branch`. See also: http://github.com/guides/remove-a-remote-branch

Several People Sharing a Single Repository

If you want to work on some stuff with other people, where you are all committing into the same repository, or even the same branch, then just share it via github.

First fork iris into your account, as from *Making Your own Copy (fork) of Iris*.

Then, go to your forked repository github page, say `http://github.com/your-user-name/iris`, select *Settings*, *Manage Access* and then *Invite collaborator*.

Note: For more information on sharing your repository see the GitHub documentation on Inviting collaborators.

Now all those people can do:

```
git clone git@githhub.com:your-user-name/iris.git
```

Remember that links starting with `git@` use the ssh protocol and are read-write; links starting with `git://` are read-only.

Your collaborators can then commit directly into that repo with the usual:

```
git commit -am 'ENH - much better code'
git push origin main  # pushes directly into your repo
```

Explore Your Repository

To see a graphical representation of the repository branches and commits:

```
gitk --all
```

To see a linear list of commits for this branch:

```
git log
```

Finally the *Fancy Log Output* `lg` alias will give you a reasonable text-based graph of the repository.

Rebasing on Trunk

For more information please see the official github documentation on git rebase.

CHAPTER

TWENTYTHREE

CONTRIBUTING TO THE DOCUMENTATION

Documentation is important and we encourage any improvements that can be made. If you believe the documentation is not clear please contribute a change to improve the documentation for all users.

Any change to the Iris project whether it is a bugfix, new feature or documentation update must use the *Development Workflow*.

23.1 Requirements

The documentation uses specific packages that need to be present. Please see *Installing Iris* for instructions.

23.2 Building

This documentation was built using the latest Python version that Iris supports. For more information see *Installing Iris*.

The build can be run from the documentation directory `docs/src`.

The build output for the html is found in the `_build/html` sub directory. When updating the documentation ensure the html build has *no errors* or *warnings* otherwise it may fail the automated cirrus-ci build.

Once the build is complete, if it is rerun it will only rebuild the impacted build artefacts so should take less time.

There is also an option to perform a build but skip the *Gallery* creation completely. This can be achieved via:

```
make html-noplot
```

If you wish to run a clean build you can run:

```
make clean
make html
```

This is useful for a final test before committing your changes.

Note: In order to preserve a clean build for the html, all **warnings** have been promoted to be **errors** to ensure they are addressed. This **only** applies when `make html` is run.

23.3 Testing

There are a ways to test various aspects of the documentation. The `make` commands shown below can be run in the `docs` or `docs/src` directory.

Each *Gallery* entry has a corresponding test. To run the tests:

```
make gallerytest
```

Many documentation pages includes python code itself that can be run to ensure it is still valid or to demonstrate examples. To ensure these tests pass run:

```
make doctest
```

See *iris.cube.Cube.data* for an example of using the doctest approach.

The hyperlinks in the documentation can be checked automatically. If there is a link that is known to work it can be excluded from the checks by adding it to the `linkcheck_ignore` array that is defined in the conf.py. The hyperlink check can be run via:

```
make linkcheck
```

If this fails check the output for the text **broken** and then correct or ignore the url.

Note: In addition to the automated cirrus-ci build of all the documentation build options above, the https://readthedocs.org/ service is also used. The configuration of this held in a file in the root of the github Iris project named `.readthedocs.yml`.

23.4 Generating API Documentation

In order to auto generate the API documentation based upon the docstrings a custom set of python scripts are used, these are located in the directory `docs/src/sphinxext`. Once the `make html` command has been run, the output of these scripts can be found in `docs/src/generated/api`.

If there is a particularly troublesome module that breaks the `make html` you can exclude the module from the API documentation. Add the entry to the `exclude_modules` tuple list in the `docs/src/sphinxext/generate_package_rst.py` file.

23.5 Gallery

The Iris *Gallery* uses a sphinx extension named sphinx-gallery that auto generates reStructuredText (rst) files based upon a gallery source directory that abides directory and filename convention.

The code for the gallery entries are in `docs/gallery_code`. Each sub directory in this directory is a sub section of the gallery. The respective `README.rst` in each folder is included in the gallery output.

For each gallery entry there must be a corresponding test script located in `docs/gallery_tests`.

To add an entry to the gallery simple place your python code into the appropriate sub directory and name it with a prefix of `plot_`. If your gallery entry does not fit into any existing sub directories then create a new directory and place it in there.

The reStructuredText (rst) output of the gallery is located in `docs/src/generated/gallery`.

For more information on the directory structure and options please see the sphinx-gallery getting started documentation.

CHAPTER

TWENTYFOUR

CONTRIBUTING TO THE CODE BASE

24.1 Code Formatting

To ensure a consistent code format throughout Iris, we recommend using tools to check the source directly.

- black for an opinionated coding auto-formatter
- flake8 linting checks

The preferred way to run these tools automatically is to setup and configure pre-commit.

You can install `pre-commit` in your development environment using `pip`:

```
$ pip install pre-commit
```

or alternatively using `conda`:

```
$ conda install -c conda-forge pre-commit
```

Note: If you have setup your Python environment using the guide *Installing from Source with Conda (Developers)* then `pre-commit` should already be present.

In order to install the `pre-commit` git hooks defined in our `.pre-commit-config.yaml` file, you must now run the following command from the root directory of Iris:

```
$ pre-commit install
```

Upon performing a `git commit`, your code will now be automatically formatted to the `black` configuration defined in our `pyproject.toml` file, and linted according to our `.flake8` configuration file. Note that, `pre-commit` will automatically download and install the necessary packages for each `.pre-commit-config.yaml` git hook.

Additionally, you may wish to enable `black` for your preferred editor/IDE.

With the `pre-commit` configured, the output of performing a `git commit` will look similar to:

```
Check for added large files..............................................Passed
Check for merge conflicts................................................Passed
Debug Statements (Python)................................(no files to check)Skipped
Don't commit to branch...................................................Passed
black....................................................(no files to check)Skipped
flake8...................................................(no files to check)Skipped
[contribution_overhaul c8513187] this is my commit message
2 files changed, 10 insertions(+), 9 deletions(-)
```

Note: You can also run black and flake8 manually. Please see the their officially documentation for more information.

24.2 Docstrings

Every public object in the Iris package should have an appropriate docstring. This is important as the docstrings are used by developers to understand the code and may be read directly in the source or via the *Iris API*.

This document has been influenced by the following PEP's,

- Attribute Docstrings **PEP 224**
- Docstring Conventions **PEP 257**

For consistency always use:

- `"""triple double quotes"""` around docstrings.
- `r"""raw triple double quotes"""` if you use any backslashes in your docstrings.
- `u"""Unicode triple-quoted string"""` for Unicode docstrings

All docstrings should be written in reST (reStructuredText) markup. See the *reST Quick Start* for more detail.

There are two forms of docstrings: **single-line** and **multi-line** docstrings.

24.2.1 Single-Line Docstrings

The single line docstring of an object must state the **purpose** of that object, known as the **purpose section**. This terse overview must be on one line and ideally no longer than 80 characters.

24.2.2 Multi-Line Docstrings

Multi-line docstrings must consist of at least a purpose section akin to the single-line docstring, followed by a blank line and then any other content, as described below. The entire docstring should be indented to the same level as the quotes at the docstring's first line.

Description

The multi-line docstring *description section* should expand on what was stated in the one line *purpose section*. The description section should try not to document *argument* and *keyword argument* details. Such information should be documented in the following *arguments and keywords section*.

Sample Multi-Line Docstring

Here is a simple example of a standard docstring:

```
def sample_routine(arg1, arg2, kwarg1="foo", kwarg2=None):
    """
    Purpose section text goes here.

    Description section longer text goes here.

    Args:

    * arg1 (numpy.ndarray):
        First argument description.
    * arg2 (numpy.ndarray):
        Second argument description.

    Kwargs:

    * kwarg1 (string):
        The first keyword argument. This argument description
        can be multi-lined.
    * kwarg2 (Boolean or None):
        The second keyword argument.

    Returns:
        numpy.ndarray of arg1 * arg2

    """
    pass
```

This would be rendered as:

documenting.docstrings_sample_routine.**sample_routine**(*arg1*, *arg2*, *kwarg1='foo'*, *kwarg2=None*)

Purpose section text goes here.

Description section longer text goes here.

Args:

- **arg1 (numpy.ndarray):** First argument description.
- **arg2 (numpy.ndarray):** Second argument description.

Kwargs:

- **kwarg1 (string):** The first keyword argument. This argument description can be multi-lined.
- **kwarg2 (Boolean or None):** The second keyword argument.

Returns numpy.ndarray of arg1 * arg2

Additionally, a summary can be extracted automatically, which would result in:

documenting.docstrings_sample_routine.sample_routine(...)	Purpose section text goes here.

24.2.3 Documenting Classes

The class constructor should be documented in the docstring for its `__init__` or `__new__` method. Methods should be documented by their own docstring, not in the class header itself.

If a class subclasses another class and its behaviour is mostly inherited from that class, its docstring should mention this and summarise the differences. Use the verb "override" to indicate that a subclass method replaces a superclass method and does not call the superclass method; use the verb "extend" to indicate that a subclass method calls the superclass method (in addition to its own behaviour).

Attribute and Property Docstrings

Here is a simple example of a class containing an attribute docstring and a property docstring:

```
class ExampleClass:
    """
    Class Summary

    """

    def __init__(self, arg1, arg2):
        """
        Purpose section description.

        Description section text.

        Args:

        * arg1 (int):
            First argument description.
        * arg2 (float):
            Second argument description.

        Returns:
            Boolean.

        """
        self.a = arg1
        "Attribute arg1 docstring."
        self.b = arg2
        "Attribute arg2 docstring."

    @property
    def square(self):
        """
        *(read-only)* Purpose section description.
```

(continues on next page)

(continued from previous page)

```
        Returns:
            int.

        """
        return self.a * self.a
```

This would be rendered as:

class documenting.docstrings_attribute.**ExampleClass**(*arg1*, *arg2*)

Purpose section description.

Description section text.

Args:

- **arg1 (int):** First argument description.
- **arg2 (float):** Second argument description.

Returns Boolean.

a

Attribute arg1 docstring.

b

Attribute arg2 docstring.

property square

(read-only) Purpose section description.

Returns int.

Note: The purpose section of the property docstring **must** state whether the property is read-only.

24.3 reST Quick Start

reST is used to create the documentation for Iris. It is used to author all of the documentation content including use in docstrings where appropriate. For more information see *Docstrings*.

reST is a lightweight markup language intended to be highly readable in source format. This guide will cover some of the more frequently used advanced reST markup syntaxes, for the basics of reST the following links may be useful:

- https://www.sphinx-doc.org/en/master/usage/restructuredtext/
- http://packages.python.org/an_example_pypi_project/sphinx.html

Reference documentation for reST can be found at http://docutils.sourceforge.net/rst.html.

24.3.1 Creating Links

Basic links can be created with `Text of the link <http://example.com>`_ which will look like Text of the link

Documents in the same project can be cross referenced with the syntax :doc:`document_name` for example, to reference the "docstrings" page :doc:`docstrings` creates the following link *Docstrings*

References can be created between sections by first making a "label" where you would like the link to point to .. _name_of_reference:: the appropriate link can now be created with :ref:`name_of_reference` (note the trailing underscore on the label)

Cross referencing other reference documentation can be achieved with the syntax :py:class:`zipfile.ZipFile` which will result in links such as `zipfile.ZipFile` and `numpy.ndarray`.

24.4 Deprecations

If you need to make a backwards-incompatible change to a public API[1] that has been included in a release (e.g. deleting a method), then you must first deprecate the old behaviour in at least one release, before removing/updating it in the next major release.

24.4.1 Adding a Deprecation

Removing a Public API

The simplest form of deprecation occurs when you need to remove a public API. The public API in question is deprecated for a period before it is removed to allow time for user code to be updated. Sometimes the deprecation is accompanied by the introduction of a new public API.

Under these circumstances the following points apply:

- Using the deprecated API must result in a concise deprecation warning which is an instance of *iris.IrisDeprecation*. It is easiest to call `iris._deprecation.warn_deprecated()`, which is a simple wrapper to `warnings.warn()` with the signature *warn_deprecation(message, **kwargs)*.
- Where possible, your deprecation warning should include advice on how to avoid using the deprecated API. For example, you might reference a preferred API, or more detailed documentation elsewhere.
- You must update the docstring for the deprecated API to include a Sphinx deprecation directive:

```
.. deprecated::  <VERSION>
```

[1] A name without a leading underscore in any of its components, with the exception of the *iris.experimental* and `iris.tests` packages.

Example public names are:

- *iris.this.*
- *iris.this.that*

Example private names are:

- *iris._this*
- *iris.this._that*
- *iris._this.that*
- *iris._this._that*
- *iris.experimental.something*
- *iris.tests.get_data_path*

where you should replace <*VERSION*> with the major and minor version of Iris in which this API is first deprecated. For example: *1.8*.

As with the deprecation warning, you should include advice on how to avoid using the deprecated API within the content of this directive. Feel free to include more detail in the updated docstring than in the deprecation warning.

- You should check the documentation for references to the deprecated API and update them as appropriate.

Changing a Default

When you need to change the default behaviour of a public API the situation is slightly more complex. The recommended solution is to use the `iris.FUTURE` object. The `iris.FUTURE` object provides boolean attributes that allow user code to control at run-time the default behaviour of corresponding public APIs. When a boolean attribute is set to *False* it causes the corresponding public API to use its deprecated default behaviour. When a boolean attribute is set to *True* it causes the corresponding public API to use its new default behaviour.

The following points apply in addition to those for removing a public API:

- You should add a new boolean attribute to `iris.FUTURE` (by modifying `iris.Future`) that controls the default behaviour of the public API that needs updating. The initial state of the new boolean attribute should be *False*. You should name the new boolean attribute to indicate that setting it to *True* will select the new default behaviour.
- You should include a reference to this `iris.FUTURE` flag in your deprecation warning and corresponding Sphinx deprecation directive.

24.4.2 Removing a Deprecation

When the time comes to make a new major release you should locate any deprecated APIs within the code that satisfy the one release minimum period described previously. Locating deprecated APIs can easily be done by searching for the Sphinx deprecation directives and/or deprecation warnings.

Removing a Public API

The deprecated API should be removed and any corresponding documentation and/or example code should be removed/updated as appropriate.

Changing a Default

- You should update the initial state of the relevant boolean attribute of `iris.FUTURE` to *True*.
- You should deprecate setting the relevant boolean attribute of `iris.Future` in the same way as described in *Removing a Public API*.

24.5 Testing

24.5.1 Test Categories

There are two main categories of tests within Iris:

- *Unit Tests*
- *Integration Tests*

Ideally, all code changes should be accompanied by one or more unit tests, and by zero or more integration tests.

But if in any doubt about what tests to add or how to write them please feel free to submit a pull-request in any state and ask for assistance.

Unit Tests

Code changes should be accompanied by enough unit tests to give a high degree of confidence that the change works as expected. In addition, the unit tests can help describe the intent behind a change.

The docstring for each test module must state the unit under test. For example:

```
"""Unit tests for the `iris.experimental.raster.export_geotiff` function."""
```

All unit tests must be placed and named according to the following structure:

Classes

When testing a class all the tests must reside in the module:

```
lib/iris/tests/unit/<fully/qualified/module>/test_<ClassName>.py
```

Within this test module each tested method must have one or more corresponding test classes, for example:

- `Test_<name of public method>`
- `Test_<name of public method>__<aspect of method>`

And within those test classes, the test methods must be named according to the aspect of the tested method which they address.

Examples:

All unit tests for *iris.cube.Cube* must reside in:

```
lib/iris/tests/unit/cube/test_Cube.py
```

Within that file the tests might look something like:

```
# Tests for the Cube.xml() method.
class Test_xml(tests.IrisTest):
    def test_some_general_stuff(self):
        ...


# Tests for the Cube.xml() method, focussing on the behaviour of
# the checksums.
class Test_xml__checksum(tests.IrisTest):
    def test_checksum_ignores_masked_values(self):
```

(continues on next page)

(continued from previous page)

```
        ...


# Tests for the Cube.add_dim_coord() method.
class Test_add_dim_coord(tests.IrisTest):
    def test_normal_usage(self):
        ...

    def test_coord_already_present(self):
        ...
```

Functions

When testing a function all the tests must reside in the module:

`lib/iris/tests/unit/<fully/qualified/module>/test_<function_name>.py`

Within this test module there must be one or more test classes, for example:

- `Test`
- `TestAspectOfFunction`

And within those test classes, the test methods must be named according to the aspect of the tested function which they address.

Examples:

All unit tests for `iris.experimental.raster.export_geotiff()` must reside in:

`lib/iris/tests/unit/experimental/raster/test_export_geotiff.py`

Within that file the tests might look something like:

```
# Tests focussing on the handling of different data types.
class TestDtypeAndValues(tests.IrisTest):
    def test_int16(self):
        ...

    def test_int16_big_endian(self):
        ...


# Tests focussing on the handling of different projections.
class TestProjection(tests.IrisTest):
    def test_no_ellipsoid(self):
        ...
```

Integration Tests

Some code changes may require tests which exercise several units in order to demonstrate an important consequence of their interaction which may not be apparent when considering the units in isolation.

These tests must be placed in the `lib/iris/tests/integration` folder. Unlike unit tests, there is no fixed naming scheme for integration tests. But folders and files must be created as required to help developers locate relevant tests. It is recommended they are named according to the capabilities under test, e.g. `metadata/test_pp_preservation.py`, and not named according to the module(s) under test.

24.5.2 Graphics Tests

Iris may be used to create various forms of graphical output; to ensure the output is consistent, there are automated tests to check against known acceptable graphical output. See *Running the Tests* for more information.

At present graphical tests are used in the following areas of Iris:

- Module `iris.tests.test_plot`
- Module `iris.tests.test_quickplot`
- *Gallery* plots contained in `docs/gallery_tests`.

Challenges

Iris uses many dependencies that provide functionality, an example that applies here is matplotlib. For more information on the dependences, see *Installing Iris*. When there are updates to the matplotlib or a dependency of matplotlib, this may result in a change in the rendered graphical output. This means that there may be no changes to Iris, but due to an updated dependency any automated tests that compare a graphical output to a known acceptable output may fail. The failure may also not be visually perceived as it may be a simple pixel shift.

Testing Strategy

The Iris Cirrus-CI matrix defines multiple test runs that use different versions of Python to ensure Iris is working as expected.

To make this manageable, the `iris.tests.IrisTest_nometa.check_graphic` test routine tests against multiple alternative **acceptable** results. It does this using an image **hash** comparison technique which avoids storing reference images in the Iris repository itself.

This consists of:

- The `iris.tests.IrisTest_nometa.check_graphic` function uses a perceptual **image hash** of the outputs (see https://github.com/JohannesBuchner/imagehash) as the basis for checking test results.
- The hashes of known **acceptable** results for each test are stored in a lookup dictionary, saved to the repo file `lib/iris/tests/results/imagerepo.json` (link) .
- An actual reference image for each hash value is stored in a *separate* public repository https://github.com/SciTools/test-iris-imagehash.
- The reference images allow human-eye assessment of whether a new output is judged to be close enough to the older ones, or not.
- The utility script `iris/tests/idiff.py` automates checking, enabling the developer to easily compare proposed new **acceptable** result images against the existing accepted reference images, for each failing test.

Reviewing Failing Tests

When you find that a graphics test in the Iris testing suite has failed, following changes in Iris or the run dependencies, this is the process you should follow:

1. Create a new, empty directory to store temporary image results, at the path `lib/iris/tests/result_image_comparison` in your Iris repository checkout.
2. **In your Iris repo root directory**, run the relevant (failing) tests directly as python scripts, or by using a command such as:

   ```
   python -m unittest discover paths/to/test/files
   ```

3. In the `iris/lib/iris/tests` folder, run the command:

   ```
   python idiff.py
   ```

   This will open a window for you to visually inspect side-by-side **old**, **new** and **difference** images for each failed graphics test. Hit a button to either *accept*, *reject* or *skip* each new result.

   If the change is **accepted**:

   - the imagehash value of the new result image is added into the relevant set of 'valid result hashes' in the image result database file, `tests/results/imagerepo.json`
   - the relevant output file in `tests/result_image_comparison` is renamed according to the image hash value, as `<hash>.png`. A copy of this new PNG file must then be added into the reference image repository at https://github.com/SciTools/test-iris-imagehash (See below).

   If a change is **skipped**:

   - no further changes are made in the repo.
   - when you run `iris/tests/idiff.py` again, the skipped choice will be presented again.

   If a change is **rejected**:

   - the output image is deleted from `result_image_comparison`.
   - when you run `iris/tests/idiff.py` again, the skipped choice will not appear, unless the relevant failing test is re-run.

4. **Now re-run the tests**. The **new** result should now be recognised and the relevant test should pass. However, some tests can perform *multiple* graphics checks within a single test case function. In those cases, any failing check will prevent the following ones from being run, so a test re-run may encounter further (new) graphical test failures. If that happens, simply repeat the check-and-accept process until all tests pass.

Add Your Changes to Iris

To add your changes to Iris, you need to make two pull requests (PR).

1. The first PR is made in the `test-iris-imagehash` repository, at https://github.com/SciTools/test-iris-imagehash.

   - First, add all the newly-generated referenced PNG files into the `images/v4` directory. In your Iris repo, these files are to be found in the temporary results folder `iris/tests/result_image_comparison`.
   - Then, to update the file which lists available images, `v4_files_listing.txt`, run from the project root directory:

```
python recreate_v4_files_listing.py
```

- Create a PR proposing these changes, in the usual way.

2. The second PR is created in the Iris repository, and should only include the change to the image results database, `tests/results/imagerepo.json`. The description box of this pull request should contain a reference to the matching one in `test-iris-imagehash`.

Note: The `result_image_comparison` folder is covered by a project `.gitignore` setting, so those files *will not show up* in a `git status` check.

Important: The Iris pull-request will not test successfully in Cirrus-CI until the `test-iris-imagehash` pull request has been merged. This is because there is an Iris test which ensures the existence of the reference images (uris) for all the targets in the image results database. It will also fail if you forgot to run `recreate_v4_files_listing.py` to update the image-listing file in `test-iris-imagehash`.

24.5.3 Running the Tests

Using setuptools for Testing Iris

Warning: The setuptools `test` command was deprecated in v41.5.0. See *Using Nox for Testing Iris*.

A prerequisite of running the tests is to have the Python environment setup. For more information on this see *Installing from Source with Conda (Developers)*.

Many Iris tests will use data that may be defined in the test itself, however this is not always the case as sometimes example files may be used. Due to the size of some of the files used these are not kept in the Iris repository. A separate repository under the SciTools organisation is used, see https://github.com/SciTools/iris-test-data.

In order to run the tests with **all** the test data you must clone the `iris-test-data` repository and then configure your shell to ensure the Iris tests can find it by using the shell environment variable named **OVERRIDE_TEST_DATA_REPOSITORY**. The example command below uses `~/projects` as the parent directory:

```
cd ~/projects
git clone git@github.com:SciTools/iris-test-data.git
export OVERRIDE_TEST_DATA_REPOSITORY=~/projects/iris-test-data/test_data
```

All the Iris tests may be run from the root `iris` project directory via:

```
python setup.py test
```

You can also run a specific test, the example below runs the tests for mapping:

```
cd lib/iris/tests
python test_mapping.py
```

When running the test directly as above you can view the command line options using the commands `python test_mapping.py -h` or `python test_mapping.py --help`.

Tip: A useful command line option to use is `-d`. This will display matplotlib figures as the tests are run. For example:

```
python test_mapping.py -d
```

You can also use the `-d` command line option when running all the tests but this will take a while to run and will require the manual closing of each of the figures for the tests to continue.

The output from running the tests is verbose as it will run ~5000 separate tests. Below is a trimmed example of the output:

```
running test
Running test suite(s): default

Running test discovery on iris.tests with 2 processors.
test_circular_subset (iris.tests.experimental.regrid.test_regrid_area_weighted_
↪rectilinear_src_and_grid.TestAreaWeightedRegrid) ... ok
test_cross_section (iris.tests.experimental.regrid.test_regrid_area_weighted_rectilinear_
↪src_and_grid.TestAreaWeightedRegrid) ... ok
test_different_cs (iris.tests.experimental.regrid.test_regrid_area_weighted_rectilinear_
↪src_and_grid.TestAreaWeightedRegrid) ... ok
...
...
test_ellipsoid (iris.tests.unit.experimental.raster.test_export_geotiff.TestProjection) .
↪.. SKIP: Test requires 'gdal'.
test_no_ellipsoid (iris.tests.unit.experimental.raster.test_export_geotiff.
↪TestProjection) ... SKIP: Test requires 'gdal'.
...
...
test_slice (iris.tests.test_util.TestAsCompatibleShape) ... ok
test_slice_and_transpose (iris.tests.test_util.TestAsCompatibleShape) ... ok
test_transpose (iris.tests.test_util.TestAsCompatibleShape) ... ok

----------------------------------------------------------------------
Ran 4762 tests in 238.649s

OK (SKIP=22)
```

There may be some tests that have been **skipped**. This is due to a Python decorator being present in the test script that will intentionally skip a test if a certain condition is not met. In the example output above there are **22** skipped tests, at the point in time when this was run this was primarily due to an experimental dependency not being present.

Tip: The most common reason for tests to be skipped is when the directory for the `iris-test-data` has not been set which would shows output such as:

```
test_coord_coord_map (iris.tests.test_plot.Test1dScatter) ... SKIP: Test(s) require␣
↪external data.
test_coord_coord (iris.tests.test_plot.Test1dScatter) ... SKIP: Test(s) require external␣
↪data.
test_coord_cube (iris.tests.test_plot.Test1dScatter) ... SKIP: Test(s) require external␣
↪data.
```

All Python decorators that skip tests will be defined in `lib/iris/tests/__init__.py` with a function name with a

prefix of `skip_`.

Using Nox for Testing Iris

Iris has adopted the use of the nox tool for automated testing on cirrus-ci and also locally on the command-line for developers.

nox is similar to tox, but instead leverages the expressiveness and power of a Python configuration file rather than an *.ini* style file. As with tox, nox can use virtualenv to create isolated Python environments, but in addition also supports conda as a testing environment backend.

Where is Nox Used?

Iris uses nox as a convenience to fully automate the process of executing the Iris tests, but also automates the process of:

- building the documentation and executing the doc-tests
- building the documentation gallery
- running the documentation URL link check
- linting the code-base
- ensuring the code-base style conforms to the black standard

You can perform all of these tasks manually yourself, however the onus is on you to first ensure that all of the required package dependencies are installed and available in the testing environment.

Nox has been configured to automatically do this for you, and provides a means to easily replicate the remote testing behaviour of cirrus-ci locally for the developer.

Installing Nox

We recommend installing nox using conda. To install nox in a separate conda environment:

```
conda create -n nox -c conda-forge nox
conda activate nox
```

To install nox in an existing active conda environment:

```
conda install -c conda-forge nox
```

The nox package is also available on PyPI, however nox has been configured to use the conda backend for Iris, so an installation of conda must always be available.

Testing with Nox

The nox configuration file *noxfile.py* is available in the root `iris` project directory, and defines all the nox sessions (i.e., tasks) that may be performed. nox must always be executed from the `iris` root directory.

To list the configured nox sessions for Iris:

```
nox --list
```

To run the Iris tests for all configured versions of Python:

```
nox --session tests
```

To build the Iris documentation specifically for Python 3.7:

```
nox --session doctest-3.7
```

To run all the Iris nox sessions:

```
nox
```

For further nox command-line options:

```
nox --help
```

Tip: For nox sessions that use the conda backend, you can use the `-v` or `--verbose` flag to display the nox conda environment package details and environment info. For example:

```
nox --session tests -- --verbose
```

Note: nox will cache its testing environments in the *.nox* root `iris` project directory.

24.5.4 Continuous Integration (CI) Testing

The Iris GitHub repository is configured to run checks against all its branches automatically whenever a pull-request is created, updated or merged. The checks performed are:

- *Cirrus-CI*
- *SciTools CLA Checker*
- *pre-commit CI*

Cirrus-CI

Iris unit and integration tests are an essential mechanism to ensure that the Iris code base is working as expected. *Running the Tests* may be performed manually by a developer locally. However Iris is configured to use the cirrus-ci service for automated Continuous Integration (CI) testing.

The cirrus-ci configuration file .cirrus.yml in the root of the Iris repository defines the tasks to be performed by cirrus-ci. For further details refer to the Cirrus-CI Documentation. The tasks performed during CI include:

- linting the code base and ensuring it adheres to the black format
- running the system, integration and unit tests for Iris
- ensuring the documentation gallery builds successfully
- performing all doc-tests within the code base
- checking all URL references within the code base and documentation are valid

The above cirrus-ci tasks are run automatically against all Iris branches on GitHub whenever a pull-request is submitted, updated or merged. See the Cirrus-CI Dashboard for details of recent past and active Iris jobs.

Cirrus CI Test environment

The test environment on the Cirrus-CI service is determined from the requirement files in `requirements/ci/py**.yml`. These are conda environment files that list the entire set of build, test and run requirements for Iris.

For reproducible test results, these environments are resolved for all their dependencies and stored as lock files in `requirements/ci/nox.lock`. The test environments will not resolve the dependencies each time, instead they will use the lock file to reproduce the same exact environment each time.

If you have updated the requirement yaml files with new dependencies, you will need to generate new lock files. To do this, run the command:

```
python tools/update_lockfiles.py -o requirements/ci/nox.lock requirements/ci/py*.yml
```

or simply:

```
make lockfiles
```

and add the changed lockfiles to your pull request.

New lockfiles are generated automatically each week to ensure that Iris continues to be tested against the latest available version of its dependencies. Each week the yaml files in `requirements/ci` are resolved by a GitHub Action. If the resolved environment has changed, a pull request is created with the new lock files. The CI test suite will run on this pull request and fixes for failed tests can be pushed to the `auto-update-lockfiles` branch to be included in the PR. Once a developer has pushed to this branch, the auto-update process will not run again until the PR is merged, to prevent overwriting developer commits. The auto-updater can still be invoked manually in this situation by going to the GitHub Actions page for the workflow, and manually running using the "Run Workflow" button. By default, this will also not override developer commits. To force an update, you must confirm "yes" in the "Run Worflow" prompt.

Skipping Cirrus-CI Tasks

As a developer you may wish to not run all the CI tasks when you are actively developing e.g., you are writing documentation and there is no need for linting, or long running compute intensive testing tasks to be executed.

As a convenience, it is possible to easily skip one or more tasks by setting the appropriate environment variable within the .cirrus.yml file to a **non-empty** string:

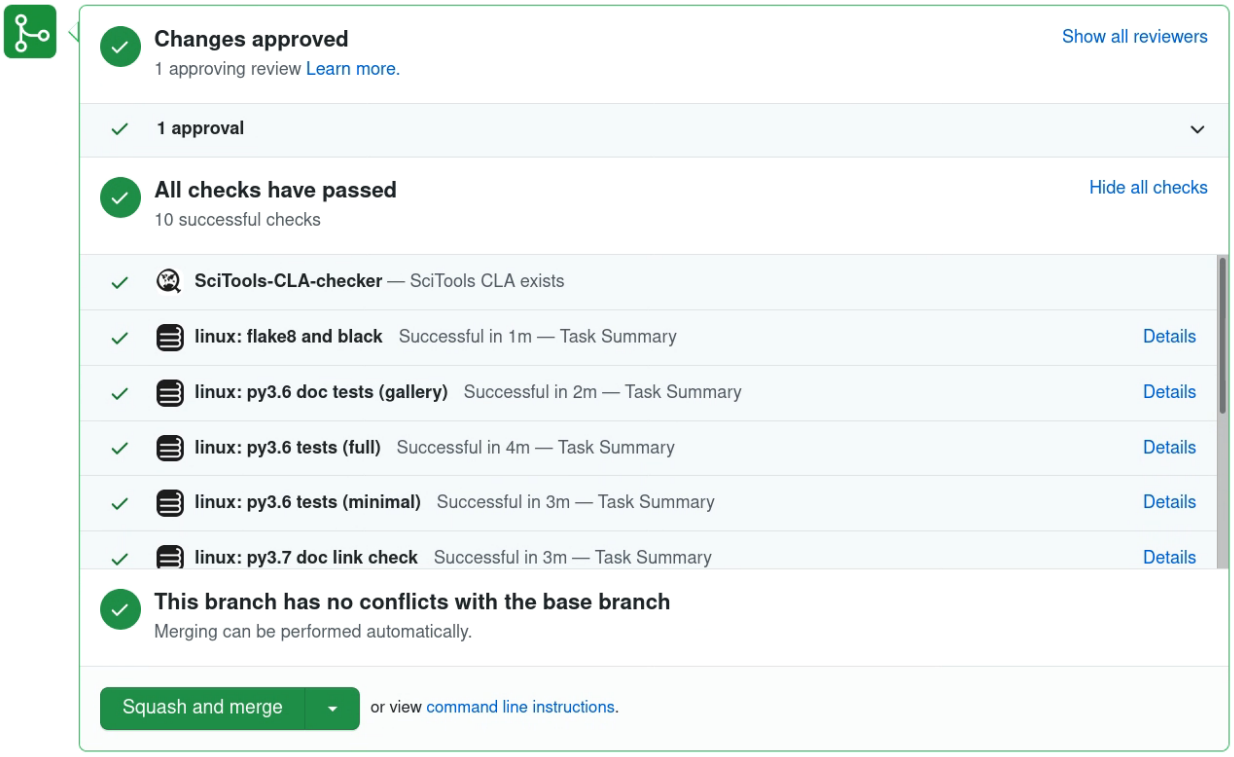
- `SKIP_LINT_TASK` to skip flake8 linting and black formatting
- `SKIP_TEST_MINIMAL_TASK` to skip restricted unit and integration testing
- `SKIP_TEST_FULL_TASK` to skip full unit and integration testing
- `SKIP_GALLERY_TASK` to skip building the documentation gallery
- `SKIP_DOCTEST_TASK` to skip running the documentation doc-tests
- `SKIP_LINKCHECK_TASK` to skip checking for broken documentation URL references
- `SKIP_ALL_TEST_TASKS` which is equivalent to setting `SKIP_TEST_MINIMAL_TASK` and `SKIP_TEST_FULL_TASK`
- `SKIP_ALL_DOC_TASKS` which is equivalent to setting `SKIP_GALLERY_TASK`, `SKIP_DOCTEST_TASK`, and `SKIP_LINKCHECK_TASK`

e.g., to skip the linting task, the following are all equivalent:

```
SKIP_LINT_TASK: "1"
SKIP_LINT_TASK: "true"
SKIP_LINT_TASK: "false"
SKIP_LINT_TASK: "skip"
SKIP_LINT_TASK: "unicorn"
```

GitHub Checklist

An example snapshot from a successful GitHub pull-request shows all tests passing:



If any CI tasks fail, then the pull-request is unlikely to be merged to the Iris target branch by a core developer.

SciTools CLA Checker

A bot which checks that the GitHub author of the pull-request has signed the **SciTools Contributor's License Agreement (CLA)**. For more information on this please see https://scitools.org.uk/organisation.html#governance.

pre-commit CI

A CI service for the pre-commit framework that checks and auto fixes all pull-requests given the Iris GitHub repository .pre-commit-config.yaml.

See the pre-commit.ci dashboard for details of recent past and active Iris jobs.

CHAPTER

TWENTYFIVE

CONTRIBUTING YOUR CHANGES

25.1 Contributing a "What's New" Entry

Iris uses a file named `latest.rst` to keep a draft of upcoming changes that will form the next release. Contributions to the *What's New in Iris* document are written by the developer most familiar with the change made. The contribution should be included as part of the Iris Pull Request that introduces the change.

The `latest.rst` and the past release notes are kept in `docs/src/whatsnew/`. If you are writing the first contribution after an Iris release: **create the new** `latest.rst` by copying the content from `latest.rst.template` in the same directory.

Since the *Contribution categories* include Internal changes, **all** Iris Pull Requests should be accompanied by a "What's New" contribution.

25.1.1 Git Conflicts

If changes to `latest.rst` are being suggested in several simultaneous Iris Pull Requests, Git will likely encounter merge conflicts. If this situation is thought likely (large PR, high repo activity etc.):

- PR author: Do not include a "What's New" entry. Mention in the PR text that a "What's New" entry is pending
- PR reviewer: Review the PR as normal. Once the PR is acceptable, ask that a **new pull request** be created specifically for the "What's New" entry, which references the main pull request and titled (e.g. for PR#9999):

  > What's New for #9999

- PR author: create the "What's New" pull request
- PR reviewer: once the "What's New" PR is created, **merge the main PR**. (this will fix any cirrus-ci linkcheck errors where the links in the "What's New" PR reference new features introduced in the main PR)
- PR reviewer: review the "What's New" PR, merge once acceptable

These measures should mean the suggested `latest.rst` changes are outstanding for the minimum time, minimising conflicts and minimising the need to rebase or merge from trunk.

25.1.2 Writing a Contribution

As introduced above, a contribution is the description of a change to Iris which improved Iris in some way. As such, a single Iris Pull Request may contain multiple changes that are worth highlighting as contributions to the what's new document.

The appropriate contribution for a pull request might in fact be an addition or change to an existing "What's New" entry.

Each contribution will ideally be written as a single concise entry using a reStructuredText auto-enumerated list `#.` directive. Where possible do not exceed **column 80** and ensure that any subsequent lines of the same entry are aligned with the first. The content should target an Iris user as the audience. The required content, in order, is as follows:

- Names of those who contributed the change. These should be their GitHub user name. Link the name to their GitHub profile. E.g. `` `@tkknight <https://github.com/tkknight>`_ changed... ``
- The new/changed behaviour
- Context to the change. Possible examples include: what this fixes, why something was added, issue references (e.g. `` :issue:`9999` ``), more specific detail on the change itself.
- Pull request references, bracketed, following the final period. E.g. `(:pull:`1111`, :pull:`9999`)`
- A trailing blank line (standard reStructuredText list format)

For example:

```
#. `@tkknight <https://github.com/tkknight>`_ changed changed argument ``x``
   to be optional in :class:`~iris.module.class` and
   :meth:`iris.module.method`. This allows greater flexibility as requested in
   :issue:`9999`. (:pull:`1111`, :pull:`9999`)
```

The above example also demonstrates some of the possible syntax for including links to code. For more inspiration on possible content and references, please examine past what's *What's New in Iris* entries.

Note: The reStructuredText syntax will be checked as part of building the documentation. Any warnings should be corrected. cirrus-ci will automatically build the documentation when creating a pull request, however you can also manually *build* the documentation.

25.1.3 Contribution Categories

The structure of the what's new release note should be easy to read by users. To achieve this several categories may be used.

Announcements General news and announcements to the Iris community.

Features Features that are new or changed to add functionality.

Bug Fixes A bug fix.

Incompatible Changes A change that causes an incompatibility with prior versions of Iris.

Deprecations Deprecations of functionality.

Dependencies Additions, removals and version changes in Iris' package dependencies.

Documentation Changes to documentation.

Internal Changes to any internal or development related topics, such as testing, environment dependencies etc.

25.2 Pull Request Checklist

All pull request will be reviewed by a core developer who will manage the process of merging. It is the responsibility of a developer submitting a pull request to do their best to deliver a pull request which meets the requirements of the project it is submitted to.

The check list summarises criteria which will be checked before a pull request is merged. Before submitting a pull request please consider this list.

1. **Provide a helpful description** of the Pull Request. This should include:

- The aim of the change / the problem addressed / a link to the issue.
- How the change has been delivered.

1. **Include a "What's New" entry**, if appropriate. See *Contributing a "What's New" Entry*.
2. **Check all tests pass**. This includes existing tests and any new tests added for any new functionality. For more information see *Running the Tests*.
3. **Check all modified and new source files conform to the required** *Code Formatting*.
4. **Check all new dependencies added to the ``requirements/ci/*.yml`` files.** If dependencies have been added then new nox testing lockfiles should be generated too, see *Cirrus CI Test environment*.
5. **Check the source documentation been updated to explain all new or changed features**. See *Docstrings*.
6. **Include code examples inside the docstrings where appropriate**. See *Testing*.
7. **Check the documentation builds without warnings or errors**. See *Building*
8. **Check for any new dependencies in the** .cirrus.yml **config file.**
9. **Check for any new dependencies in the** readthedocs.yml **file**. This file is used to build the documentation that is served from https://scitools-iris.readthedocs.io/en/latest/
10. **Check for updates needed for supporting projects for test or example data**. For example:

    - iris-test-data is a github project containing all the data to support the tests.
    - iris-sample-data is a github project containing all the data to support the gallery and examples.
    - test-iris-imagehash is a github project containing reference plot images to support Iris *Graphics Tests*.

    If new files are required by tests or code examples, they must be added to the appropriate supporting project via a suitable pull-request. This pull request should be referenced in the main Iris pull request and must be accepted and merged before the Iris one can be.

CHAPTER

TWENTYSIX

RELEASES

A release of Iris is a tag on the SciTools/Iris Github repository.

The summary below is of the main areas that constitute the release. The final section details the *Maintainer Steps* to take.

26.1 Before Release

26.1.1 Deprecations

Ensure that any behaviour which has been deprecated for the correct number of previous releases is now finally changed. More detail, including the correct number of releases, is in *Deprecations*.

26.2 Release Branch

Once the features intended for the release are on `main`, a release branch should be created, in the `SciTools/iris` repository. This will have the name:

```
v{major release number}.{minor release number}.x
```

for example:

```
v1.9.x
```

This branch shall be used to finalise the release details in preparation for the release candidate.

26.3 Release Candidate

Prior to a release, a release candidate tag may be created, marked as a pre-release in GitHub, with a tag ending with `rc` followed by a number (0-based), e.g.,:

```
v1.9.0rc0
```

If created, the pre-release shall be available for a minimum of two weeks prior to the release being cut. However a 4 week period should be the goal to allow user groups to be notified of the existence of the pre-release and encouraged to test the functionality.

A pre-release is expected for a major or minor release, but not for a point release.

If new features are required for a release after a release candidate has been cut, a new pre-release shall be issued first.

Make the release candidate available as a conda package on the conda-forge Anaconda channel using the rc_iris label. To do this visit the conda-forge iris-feedstock and follow CFEP-05. For further information see the conda-forge User Documentation.

26.4 Documentation

The documentation should include all of the `whatsnew` entries for the release. This content should be reviewed and adapted as required.

Steps to achieve this can be found in the *Maintainer Steps*.

26.5 The Release

The final steps of the release are to change the version string `__version__` in the source of `iris.__init__.py` and ensure the release date and details are correct in the relevant `whatsnew` page within the documentation.

Once all checks are complete, the release is cut by the creation of a new tag in the `SciTools/iris` repository.

26.6 Update conda-forge

Once a release is cut on GitHub, update the Iris conda recipe on the conda-forge iris-feedstock for the release. This will build and publish the conda package on the conda-forge Anaconda channel.

26.7 Update PyPI

Update the scitools-iris project on PyPI with the latest Iris release.

To do this perform the following steps.

Create a conda environment with the appropriate conda packages to build the source distribution (`sdist`) and pure Python wheel (`bdist_wheel`):

```
> conda create -n iris-pypi -c conda-forge --yes pip python setuptools twine wheel
> . activate iris-pypi
```

Checkout the appropriate Iris <release> tag from the appropriate <repo>. For example, to checkout tag `v1.0` from `upstream`:

```
> git fetch upstream --tags
> git checkout v1.0
```

Build the source distribution and wheel from the Iris root directory:

```
> python setup.py sdist bdist_wheel
```

This `./dist` directory should now be populated with the source archive `.tar.gz` file, and built distribution `.whl` file.

Check that the package description will render properly on PyPI for each of the built artifacts:

```
> python -m twine check dist/*
```

To list and check the contents of the binary wheel:

```
> python -m zipfile --list dist/*.whl
```

If all seems well, sufficient maintainer privileges will be required to upload these artifacts to scitools-iris on PyPI:

```
> python -m twine upload --repository-url https://upload.pypi.org/legacy/ dist/*
```

Ensure that the artifacts are successfully uploaded and available on scitools-iris before creating a conda test environment to install Iris from PyPI:

```
> conda deactivate
> conda env create --file ./requrements/ci/iris.yml
> . activate iris-dev
> conda install -c conda-forge pip
> python -m pip install --no-deps scitools-iris
```

For further details on how to test Iris, see *Running the Tests*.

See also:

For further information on packaging and uploading a project to PyPI, please refer to Generating Distribution Archives and Packaging Your Project.

26.8 Merge Back

After the release is cut, the changes from the release branch should be merged back onto the `SciTools/iris` `main` branch.

To achieve this, first cut a local branch from the latest `main` branch, and *git merge* the `.x` release branch into it. Ensure that the `iris.__version__`, `docs/src/whatsnew/index.rst` and `docs/src/whatsnew/latest.rst` are correct, before committing these changes and then proposing a pull-request on the `main` branch of `SciTools/iris`.

26.9 Point Releases

Bug fixes may be implemented and targeted on the `.x` release branch. These should lead to a new point release, and another tag. For example, a fix for a problem with the `v1.9.0` release will be merged into `v1.9.x` release branch, and then released by tagging `v1.9.1`.

New features shall not be included in a point release, these are for bug fixes.

A point release does not require a release candidate, but the rest of the release process is to be followed, including the merge back of changes into `main`.

26.10 Maintainer Steps

These steps assume a release for `1.9.0` is to be created.

26.10.1 Release Steps

1. Create the release feature branch `v1.9.x` on SciTools/iris. The only exception is for a point/bugfix release, as it should already exist
2. Update the `iris.__init__.py` version string e.g., to `1.9.0`
3. Update the `whatsnew` for the release:
   - Use `git` to rename `docs/src/whatsnew/latest.rst` to the release version file `v1.9.rst`
   - Use `git` to delete the `docs/src/whatsnew/latest.rst.template` file
   - In `v1.9.rst` remove the `[unreleased]` caption from the page title. Note that, the Iris version and release date are updated automatically when the documentation is built
   - Review the file for correctness
   - Work with the development team to populate the `Release Highlights` dropdown at the top of the file, which provides extra detail on notable changes
   - Use `git` to add and commit all changes, including removal of `latest.rst.template`
4. Update the `whatsnew` index `docs/src/whatsnew/index.rst`
   - Remove the reference to `latest.rst`
   - Add a reference to `v1.9.rst` to the top of the list
5. Check your changes by building the documentation and reviewing
6. Once all the above steps are complete, the release is cut, using the *Draft a new release* button on the Iris release page

26.10.2 Post Release Steps

1. Check the documentation has built on Read The Docs. The build is triggered by any commit to `main`. Additionally check that the versions available in the pop out menu in the bottom left corner include the new release version. If it is not present you will need to configure the versions available in the **admin** dashboard in Read The Docs.
2. Review the Active Versions for the `scitools-iris` project on Read The Docs to ensure that the appropriate versions are `Active` and/or `Hidden`. To do this `Edit` the appropriate version e.g., see Editing v3.0.0rc0 (must be logged into Read the Docs).
3. Copy `docs/src/whatsnew/latest.rst.template` to `docs/src/whatsnew/latest.rst`. This will reset the file with the `unreleased` heading and placeholders for the `whatsnew` headings
4. Add back in the reference to `latest.rst` to the `whatsnew` index `docs/src/whatsnew/index.rst`
5. Update `iris.__init__.py` version string to show as `1.10.dev0`
6. Merge back to `main`

CHAPTER

TWENTYSEVEN

IRIS API

27.1 iris.analysis

27.1.1 iris.analysis.calculus

Calculus operations on *iris.cube.Cube* instances.

See also: NumPy.

In this module:

- *cube_delta*
- *differentiate*
- *curl*

iris.analysis.calculus.**cube_delta**(*cube*, *coord*)

Given a cube calculate the difference between each value in the given coord's direction.

Args:

- **coord** either a Coord instance or the unique name of a coordinate in the cube. If a Coord instance is provided, it does not necessarily have to exist in the cube.

Example usage:

```
change_in_temperature_wrt_pressure = cube_delta(temperature_cube,
↪'pressure')
```

Note: Missing data support not yet implemented.

---

iris.analysis.calculus.**differentiate**(*cube*, *coord_to_differentiate*)

Calculate the differential of a given cube with respect to the coord_to_differentiate.

Args:

- **coord_to_differentiate:** Either a Coord instance or the unique name of a coordinate which exists in the cube. If a Coord instance is provided, it does not necessarily have to exist on the cube.

Example usage:

```
u_wind_acceleration = differentiate(u_wind_cube, 'forecast_time')
```

The algorithm used is equivalent to:

$$d_i = \frac{v_{i+1} - v_i}{c_{i+1} - c_i}$$

Where `d` is the differential, `v` is the data value, `c` is the coordinate value and `i` is the index in the differential direction. Hence, in a normal situation if a cube has a shape (x: n; y: m) differentiating with respect to x will result in a cube of shape (x: n-1; y: m) and differentiating with respect to y will result in (x: n; y: m-1). If the coordinate to differentiate is *circular* then the resultant shape will be the same as the input cube.

In the returned cube the *coord_to_differentiate* object is redefined such that the output coordinate values are set to the averages of the original coordinate values (i.e. the midpoints). Similarly, the output lower bounds values are set to the averages of the original lower bounds values and the output upper bounds values are set to the averages of the original upper bounds values. In more formal terms:

- *C[i] = (c[i] + c[i+1]) / 2*
- *B[i, 0] = (b[i, 0] + b[i+1, 0]) / 2*
- *B[i, 1] = (b[i, 1] + b[i+1, 1]) / 2*

where *c* and *b* represent the input coordinate values and bounds, and *C* and *B* the output coordinate values and bounds.

Note: Difference method used is the same as *cube_delta()* and therefore has the same limitations.

Note: Spherical differentiation does not occur in this routine.

`iris.analysis.calculus.curl(i_cube, j_cube, k_cube=None)`

Calculate the 2-dimensional or 3-dimensional spherical or cartesian curl of the given vector of cubes.

As well as the standard x and y coordinates, this function requires each cube to possess a vertical or z-like coordinate (representing some form of height or pressure). This can be a scalar or dimension coordinate.

Args:

- **i_cube** The i cube of the vector to operate on
- **j_cube** The j cube of the vector to operate on

Kwargs:

- **k_cube** The k cube of the vector to operate on

Return (i_cmpt_curl_cube, j_cmpt_curl_cube, k_cmpt_curl_cube)

If the k-cube is not passed in then the 2-dimensional curl will be calculated, yielding the result: [None, None, k_cube]. If the k-cube is passed in, the 3-dimensional curl will be calculated, returning 3 component cubes.

All cubes passed in must have the same data units, and those units must be spatially-derived (e.g. 'm/s' or 'km/h').

The calculation of curl is dependent on the type of *CoordSystem()* in the cube. If the *CoordSystem()* is either GeogCS or RotatedGeogCS, the spherical curl will be calculated; otherwise the cartesian curl will be calculated:

Cartesian curl

When cartesian calculus is used, i_cube is the u component, j_cube is the v component and k_cube is the w component.

The Cartesian curl is defined as:

$$\nabla \times \vec{u} = (\frac{\delta w}{\delta y} - \frac{\delta v}{\delta z})\vec{a}_i - (\frac{\delta w}{\delta x} - \frac{\delta u}{\delta z})\vec{a}_j + (\frac{\delta v}{\delta x} - \frac{\delta u}{\delta y})\vec{a}_k$$

Spherical curl
When spherical calculus is used, i_cube is the ϕ vector component (e.g. eastward), j_cube is the θ component (e.g. northward) and k_cube is the radial component.

The spherical curl is defined as:

$$\nabla \times \vec{A} = \frac{1}{rcos\theta}(\frac{\delta}{\delta\theta}(\vec{A}_\phi cos\theta) - \frac{\delta\vec{A}_\theta}{\delta\phi})\vec{r} + \frac{1}{r}(\frac{1}{cos\theta}\frac{\delta\vec{A}_r}{\delta\phi} - \frac{\delta}{\delta r}(r\vec{A}_\phi))\vec{\theta} + \frac{1}{r}(\frac{\delta}{\delta r}(r\vec{A}_\theta) - \frac{\delta\vec{A}_r}{\delta\theta})\vec{\phi}$$

where phi is longitude, theta is latitude.

27.1.2 iris.analysis.cartography

Various utilities and numeric transformations relevant to cartography.

In this module:

- *area_weights*
- *cosine_latitude_weights*
- *get_xy_contiguous_bounded_grids*
- *get_xy_grids*
- *gridcell_angles*
- *project*
- *rotate_grid_vectors*
- *rotate_pole*
- *rotate_winds*
- *unrotate_pole*
- *wrap_lons*
- *DistanceDifferential*
- *PartialDifferential*

iris.analysis.cartography.**area_weights**(*cube*, *normalize=False*)
Returns an array of area weights, with the same dimensions as the cube.

This is a 2D lat/lon area weights array, repeated over the non lat/lon dimensions.

Args:
- **cube (*iris.cube.Cube*):** The cube to calculate area weights for.

Kwargs:
- **normalize (False/True):** If False, weights are grid cell areas. If True, weights are grid cell areas divided by the total grid area.

The cube must have coordinates 'latitude' and 'longitude' with bounds.

Area weights are calculated for each lat/lon cell as:

$$r^2(lon_1 - lon_0)(\sin(lat_1) - \sin(lat_0))$$

Currently, only supports a spherical datum. Uses earth radius from the cube, if present and spherical. Defaults to iris.analysis.cartography.DEFAULT_SPHERICAL_EARTH_RADIUS.

---

iris.analysis.cartography.**cosine_latitude_weights**(*cube*)

Returns an array of latitude weights, with the same dimensions as the cube. The weights are the cosine of latitude.

These are n-dimensional latitude weights repeated over the dimensions not covered by the latitude coordinate.

The cube must have a coordinate with 'latitude' in the name. Out of range values (greater than 90 degrees or less than -90 degrees) will be clipped to the valid range.

Weights are calculated for each latitude as:

$$w_l = \cos \phi_l$$

Examples:

Compute weights suitable for averaging type operations:

```
from iris.analysis.cartography import cosine_latitude_weights
cube = iris.load_cube(iris.sample_data_path('air_temp.pp'))
weights = cosine_latitude_weights(cube)
```

Compute weights suitable for EOF analysis (or other covariance type analyses):

```
import numpy as np
from iris.analysis.cartography import cosine_latitude_weights
cube = iris.load_cube(iris.sample_data_path('air_temp.pp'))
weights = np.sqrt(cosine_latitude_weights(cube))
```

---

iris.analysis.cartography.**get_xy_contiguous_bounded_grids**(*cube*)

Return 2d arrays for x and y bounds.

Returns array of shape (n+1, m+1).

Example:

```
xs, ys = get_xy_contiguous_bounded_grids(cube)
```

---

iris.analysis.cartography.**get_xy_grids**(*cube*)

Return 2D X and Y points for a given cube.

Parameters points. (** cube - The cube for which to generate 2D X and Y*) –

Example:

```
x, y = get_xy_grids(cube)
```

iris.analysis.cartography.**gridcell_angles**(*x*, *y=None*, *cell_angle_boundpoints='mid-lhs, mid-rhs'*)

Calculate gridcell orientations for an arbitrary 2-dimensional grid.

The input grid is defined by two 2-dimensional coordinate arrays with the same dimensions (ny, nx), specifying the geolocations of a 2D mesh.

Input values may be coordinate points (ny, nx) or bounds (ny, nx, 4). However, if points, the edges in the X direction are assumed to be connected by wraparound.

Input can be either two arrays, two coordinates, or a single cube containing two suitable coordinates identified with the 'x' and 'y' axes.

Args:

The inputs (x [,y]) can be any of the following :

- **x (*Cube*):** a grid cube with 2D X and Y coordinates, identified by 'axis'. The coordinates must be 2-dimensional with the same shape. The two dimensions represent grid dimensions in the order Y, then X.
- **x, y (*Coord*):** X and Y coordinates, specifying grid locations on the globe. The coordinates must be 2-dimensional with the same shape. The two dimensions represent grid dimensions in the order Y, then X. If there is no coordinate system, they are assumed to be true longitudes and latitudes. Units must convertible to 'degrees'.
- **x, y (2-dimensional arrays of same shape (ny, nx)):** longitude and latitude cell center locations, in degrees. The two dimensions represent grid dimensions in the order Y, then X.
- **x, y (3-dimensional arrays of same shape (ny, nx, 4)):** longitude and latitude cell bounds, in degrees. The first two dimensions are grid dimensions in the order Y, then X. The last index maps cell corners anticlockwise from bottom-left.

Optional Args:

- **cell_angle_boundpoints (string):** Controls which gridcell bounds locations are used to calculate angles, if the inputs are bounds or bounded coordinates. Valid values are 'lower-left, lower-right', which takes the angle from the lower left to the lower right corner, and 'mid-lhs, mid-rhs' which takes an angles between the average of the left-hand and right-hand pairs of corners. The default is 'mid-lhs, mid-rhs'.

Returns

(2-dimensional cube)
Cube of angles of grid-x vector from true Eastward direction for each grid-cell, in degrees. It also has "true" longitude and latitude coordinates, with no coordinate system. When the input has coords, then the output ones are identical if the inputs are true-latlons, otherwise they are transformed true-latlon versions. When the input has bounded coords, then the output coords have matching bounds and centrepoints (possibly transformed). When the input is 2d arrays, or has unbounded coords, then the output coords have matching points and no bounds. When the input is 3d arrays, then the output coords have matching bounds, and the centrepoints are an average of the 4 boundpoints.

Return type angles

iris.analysis.cartography.**project**(*cube*, *target_proj*, *nx=None*, *ny=None*)

Nearest neighbour regrid to a specified target projection.

Return a new cube that is the result of projecting a cube with 1 or 2 dimensional latitude-longitude coordinates from its coordinate system into a specified projection e.g. Robinson or Polar Stereographic. This function is intended to be used in cases where the cube's coordinates prevent one from directly visualising the data, e.g. when the longitude and latitude are two dimensional and do not make up a regular grid.

Parameters

- **cube** (*) – An instance of *iris.cube.Cube*.
- **target_proj** (*) – An instance of the Cartopy Projection class, or an instance of *iris.coord_systems.CoordSystem* from which a projection will be obtained.

Kwargs:

- **nx** Desired number of sample points in the x direction for a domain covering the globe.
- **ny** Desired number of sample points in the y direction for a domain covering the globe.

Returns An instance of *iris.cube.Cube* and a list describing the extent of the projection.

Note: This function assumes global data and will if necessary extrapolate beyond the geographical extent of the source cube using a nearest neighbour approach. nx and ny then include those points which are outside of the target projection.

Note: Masked arrays are handled by passing their masked status to the resulting nearest neighbour values. If masked, the value in the resulting cube is set to 0.

Warning: This function uses a nearest neighbour approach rather than any form of linear/non-linear interpolation to determine the data value of each cell in the resulting cube. Consequently it may have an adverse effect on the statistics of the data e.g. the mean and standard deviation will not be preserved.

Warning: If the target projection is non-rectangular, e.g. Robinson, the target grid may include points outside the boundary of the projection. The latitude/longitude of such points may be unpredictable.

---

iris.analysis.cartography.**rotate_grid_vectors**(*u_cube*, *v_cube*, *grid_angles_cube=None*, *grid_angles_kwargs=None*)

Rotate distance vectors from grid-oriented to true-latlon-oriented.

Can also rotate by arbitrary angles, if they are passed in.

Note: This operation overlaps somewhat in function with *iris.analysis.cartography.rotate_winds()*. However, that routine only rotates vectors according to

transformations between coordinate systems. This function, by contrast, can rotate vectors by arbitrary angles. Most commonly, the angles are estimated solely from grid sampling points, using *gridcell_angles()* : This allows operation on complex meshes defined by two-dimensional coordinates, such as most ocean grids.

Args:

- **u_cube, v_cube** [(cube)] Cubes of grid-u and grid-v vector components. Units should be differentials of true-distance, e.g. 'm/s'.

Optional args:

- **grid_angles_cube** [(cube)] gridcell orientation angles. Units must be angular, i.e. can be converted to 'radians'. If not provided, grid angles are estimated from 'u_cube' using the *gridcell_angles()* method.
- **grid_angles_kwargs** [(dict or None)] Additional keyword args to be passed to the *gridcell_angles()* method, if it is used.

Returns

(cube) Cubes of true-north oriented vector components. Units are same as inputs.

Note: Vector magnitudes will always be the same as the inputs.

Return type true_u, true_v

---

iris.analysis.cartography.**rotate_pole**(*lons*, *lats*, *pole_lon*, *pole_lat*)

Convert arrays of longitudes and latitudes to arrays of rotated-pole longitudes and latitudes. The values of `pole_lon` and `pole_lat` should describe the rotated pole that the arrays of longitudes and latitudes are to be rotated onto.

As the arrays of longitudes and latitudes must describe a rectilinear grid, the arrays of rotated-pole longitudes and latitudes must be of the same shape as each other.

Example:

```
rotated_lons, rotated_lats = rotate_pole(lons, lats,         pole_
↪lon, pole_lat)
```

Note: Uses proj.4 to perform the conversion.

Parameters

- **lons** (*) – An array of longitude values.
- **lats** (*) – An array of latitude values.
- **pole_lon** (*) – The longitude of the rotated pole that the arrays of longitudes and latitudes are to be rotated onto.
- **pole_lat** (*) – The latitude of the rotated pole that the arrays of longitudes and latitudes are to be rotated onto.

Returns An array of rotated-pole longitudes and an array of rotated-pole latitudes.

---

iris.analysis.cartography.**rotate_winds**(*u_cube*, *v_cube*, *target_cs*)

Transform wind vectors to a different coordinate system.

The input cubes contain U and V components parallel to the local X and Y directions of the input grid at each point.

The output cubes contain the same winds, at the same locations, but relative to the grid directions of a different coordinate system. Thus in vector terms, the magnitudes will always be the same, but the angles can be different.

The outputs retain the original horizontal dimension coordinates, but also have two 2-dimensional auxiliary coordinates containing the X and Y locations in the target coordinate system.

Args:

- **u_cube** An instance of *iris.cube.Cube* that contains the x-component of the vector.
- **v_cube** An instance of *iris.cube.Cube* that contains the y-component of the vector.
- **target_cs** An instance of *iris.coord_systems.CoordSystem* that specifies the new grid directions.

Returns A (u', v') tuple of *iris.cube.Cube* instances that are the u and v components in the requested target coordinate system. The units are the same as the inputs.

Note: The U and V values relate to distance, with units such as 'm s-1'. These are not the same as coordinate vectors, which transform in a different manner.

Note: The names of the output cubes are those of the inputs, prefixed with 'transformed_' (e.g. 'transformed_x_wind').

Warning: Conversion between rotated-pole and non-rotated systems can be expressed analytically. However, this function always uses a numerical approach. In locations where this numerical approach does not preserve magnitude to an accuracy of 0.1%, the corresponding elements of the returned cubes will be masked.

---

iris.analysis.cartography.**unrotate_pole**(*rotated_lons*, *rotated_lats*, *pole_lon*, *pole_lat*)

Convert arrays of rotated-pole longitudes and latitudes to unrotated arrays of longitudes and latitudes. The values of `pole_lon` and `pole_lat` should describe the location of the rotated pole that describes the arrays of rotated-pole longitudes and latitudes.

As the arrays of rotated-pole longitudes and latitudes must describe a rectilinear grid, the arrays of rotated-pole longitudes and latitudes must be of the same shape as each other.

Example:

```
lons, lats = unrotate_pole(rotated_lons, rotated_lats,
↪pole_lon, pole_lat)
```

Note: Uses proj.4 to perform the conversion.

Parameters

- **rotated_lons** (*) – An array of rotated-pole longitude values.

- **rotated_lats** (*) – An array of rotated-pole latitude values.
- **pole_lon** (*) – The longitude of the rotated pole that describes the arrays of rotated-pole longitudes and latitudes.
- **pole_lat** (*) – The latitude of the rotated pole that describes the arrays of rotated-pole longitudes and latitudes.

Returns An array of unrotated longitudes and an array of unrotated latitudes.

---

iris.analysis.cartography.**wrap_lons**(*lons*, *base*, *period*)

Wrap longitude values into the range between base and base+period.

For example:

```
>>> print(wrap_lons(np.array([185, 30, -200, 75]), -180, 360))
[-175.   30.  160.   75.]
```

DistanceDifferential(dx1, dy1, dx2, dy2)

class iris.analysis.cartography.**DistanceDifferential**(*_cls*, *dx1*, *dy1*, *dx2*, *dy2*)

Create new instance of DistanceDifferential(dx1, dy1, dx2, dy2)

count(*value*, */*)
Return number of occurrences of value.

index(*value*, *start=0*, *stop=9223372036854775807*, */*)
Return first index of value.

Raises ValueError if the value is not present.

dx1
Alias for field number 0

dx2
Alias for field number 2

dy1
Alias for field number 1

dy2
Alias for field number 3

PartialDifferential(dx1, dy1)

class iris.analysis.cartography.**PartialDifferential**(*_cls*, *dx1*, *dy1*)

Create new instance of PartialDifferential(dx1, dy1)

count(*value*, */*)
Return number of occurrences of value.

index(*value*, *start=0*, *stop=9223372036854775807*, */*)
Return first index of value.

Raises ValueError if the value is not present.

dx1
Alias for field number 0

dy1
Alias for field number 1

27.1.3 iris.analysis.geometry

Various utilities related to geometric operations.

Note: This module requires `shapely`.

In this module:

- *geometry_area_weights*

`iris.analysis.geometry.`**`geometry_area_weights`**(*cube*, *geometry*, *normalize=False*)
Returns the array of weights corresponding to the area of overlap between the cells of cube's horizontal grid, and the given shapely geometry.

The returned array is suitable for use with *iris.analysis.MEAN*.

The cube must have bounded horizontal coordinates.

Note: This routine works in Euclidean space. Area calculations do not account for the curvature of the Earth. And care must be taken to ensure any longitude values are expressed over a suitable interval.

Note: This routine currently does not handle all out-of-bounds cases correctly. In cases where both the coordinate bounds and the geometry's bounds lie outside the physically realistic range (i.e., abs(latitude) > 90., as it is commonly the case when bounds are constructed via guess_bounds()), the weights calculation might be wrong. In this case, a UserWarning will be issued.

Args:

- **cube (*iris.cube.Cube*):** A Cube containing a bounded, horizontal grid definition.
- **geometry (a shapely geometry instance):** The geometry of interest. To produce meaningful results this geometry must have a non-zero area. Typically a Polygon or MultiPolygon.

Kwargs:

- **normalize:** Calculate each individual cell weight as the cell area overlap between the cell and the given shapely geometry divided by the total cell area. Default is False.

27.1.4 iris.analysis.maths

Basic mathematical and statistical operations.

In this module:

- *abs*
- *add*
- *apply_ufunc*
- *divide*
- *exp*
- *exponentiate*

- *intersection_of_cubes*
- *log*
- *log10*
- *log2*
- *multiply*
- *subtract*
- *IFunc*

iris.analysis.maths.**abs**(*cube*, *in_place=False*)

Calculate the absolute values of the data in the Cube provided.

Args:
- **cube:** An instance of *iris.cube.Cube*.

Kwargs:
- **in_place:** Whether to create a new Cube, or alter the given "cube".

Returns An instance of *iris.cube.Cube*.

---

iris.analysis.maths.**add**(*cube*, *other*, *dim=None*, *in_place=False*)

Calculate the sum of two cubes, or the sum of a cube and a coordinate or scalar value.

When summing two cubes, they must both have the same coordinate systems & data resolution.

When adding a coordinate to a cube, they must both share the same number of elements along a shared axis.

Args:
- **cube:** An instance of *iris.cube.Cube*.
- **other:** An instance of *iris.cube.Cube* or *iris.coords.Coord*, or a number or numpy.ndarray.

Kwargs:
- **dim:** If supplying a coord with no match on the cube, you must supply the dimension to process.
- **in_place:** Whether to create a new Cube, or alter the given "cube".

Returns An instance of *iris.cube.Cube*.

---

iris.analysis.maths.**apply_ufunc**(*ufunc*, *cube*, *other=None*, *new_unit=None*, *new_name=None*, *in_place=False*)

Apply a numpy universal function to a cube or pair of cubes.

Note: Many of the numpy.ufunc have been implemented explicitly in Iris e.g. `numpy.abs()`, `numpy.add()` are implemented in *iris.analysis.maths.abs()*, *iris.analysis.maths.add()*. It is usually preferable to use these functions rather than *iris.analysis.maths.apply_ufunc()* where possible.

Args:
- **ufunc:** An instance of `numpy.ufunc()` e.g. `numpy.sin()`, `numpy.mod()`.
- **cube:** An instance of *iris.cube.Cube*.

Kwargs:

- **other:** An instance of *iris.cube.Cube* to be given as the second argument to numpy.ufunc().
- **new_unit:** Unit for the resulting Cube.
- **new_name:** Name for the resulting Cube.
- **in_place:** Whether to create a new Cube, or alter the given "cube".

Returns An instance of *iris.cube.Cube*.

Example:

```
cube = apply_ufunc(numpy.sin, cube, in_place=True)
```

---

iris.analysis.maths.**divide**(*cube*, *other*, *dim=None*, *in_place=False*)

Calculate the division of a cube by a cube or coordinate.

Args:
- **cube:** An instance of *iris.cube.Cube*.
- **other:** An instance of *iris.cube.Cube* or *iris.coords.Coord*, or a number or numpy.ndarray.

Kwargs:
- **dim:** If supplying a coord with no match on the cube, you must supply the dimension to process.

Returns An instance of *iris.cube.Cube*.

---

iris.analysis.maths.**exp**(*cube*, *in_place=False*)

Calculate the exponential (exp(x)) of the cube.

Args:
- **cube:** An instance of *iris.cube.Cube*.

Note: Taking an exponential will return a cube with dimensionless units.

Kwargs:
- **in_place:** Whether to create a new Cube, or alter the given "cube".

Returns An instance of *iris.cube.Cube*.

---

iris.analysis.maths.**exponentiate**(*cube*, *exponent*, *in_place=False*)

Returns the result of the given cube to the power of a scalar.

Args:
- **cube:** An instance of *iris.cube.Cube*.
- **exponent:** The integer or floating point exponent.

Note: When applied to the cube's unit, the exponent must result in a unit that can be described using only integer powers of the basic units.

e.g. Unit('meter^-2 kilogram second^-1')

Kwargs:
- **in_place:** Whether to create a new Cube, or alter the given "cube".

Returns An instance of *iris.cube.Cube*.

iris.analysis.maths.**intersection_of_cubes**(*cube*, *other_cube*)
Return the two Cubes of intersection given two Cubes.

Note: The intersection of cubes function will ignore all single valued coordinates in checking the intersection.

Args:
- **cube:** An instance of *iris.cube.Cube*.
- **other_cube:** An instance of *iris.cube.Cube*.

Returns A pair of *iris.cube.Cube* instances in a tuple corresponding to the original cubes restricted to their intersection.

---

iris.analysis.maths.**log**(*cube*, *in_place=False*)
Calculate the natural logarithm (base-e logarithm) of the cube.

Args:
- **cube:** An instance of *iris.cube.Cube*.

Kwargs:
- **in_place:** Whether to create a new Cube, or alter the given "cube".

Returns An instance of *iris.cube.Cube*.

---

iris.analysis.maths.**log10**(*cube*, *in_place=False*)
Calculate the base-10 logarithm of the cube.

Args:
- **cube:** An instance of *iris.cube.Cube*.

Kwargs:
- **in_place:** Whether to create a new Cube, or alter the given "cube".

Returns An instance of *iris.cube.Cube*.

---

iris.analysis.maths.**log2**(*cube*, *in_place=False*)
Calculate the base-2 logarithm of the cube.

Args:
- **cube:** An instance of *iris.cube.Cube*.

Kwargs:lib/iris/tests/unit/analysis/maths/test_subtract.py
- **in_place:** Whether to create a new Cube, or alter the given "cube".

Returns An instance of *iris.cube.Cube*.

---

iris.analysis.maths.**multiply**(*cube*, *other*, *dim=None*, *in_place=False*)
Calculate the product of a cube and another cube or coordinate.

Args:
- **cube:** An instance of *iris.cube.Cube*.
- **other:** An instance of *iris.cube.Cube* or *iris.coords.Coord*, or a number or *numpy.ndarray*.

Kwargs:
- **dim:** If supplying a coord with no match on the cube, you must supply the dimension to process.

Returns An instance of *iris.cube.Cube*.

---

iris.analysis.maths.**subtract**(*cube*, *other*, *dim=None*, *in_place=False*)

Calculate the difference between two cubes, or the difference between a cube and a coordinate or scalar value.

When subtracting two cubes, they must both have the same coordinate systems & data resolution.

When subtracting a coordinate to a cube, they must both share the same number of elements along a shared axis.

Args:
- **cube:** An instance of *iris.cube.Cube*.
- **other:** An instance of *iris.cube.Cube* or *iris.coords.Coord*, or a number or numpy.ndarray.

Kwargs:
- **dim:** If supplying a coord with no match on the cube, you must supply the dimension to process.
- **in_place:** Whether to create a new Cube, or alter the given "cube".

Returns An instance of *iris.cube.Cube*.

IFunc class for functions that can be applied to an iris cube.

class iris.analysis.maths.**IFunc**(*data_func*, *units_func*)

Create an ifunc from a data function and units function.

Args:
- data_func:
  Function to be applied to one or two data arrays, which are given as positional arguments. Should return another data array, with the same shape as the first array.

  May also have keyword arguments.
- units_func:
  Function to calculate the units of the resulting cube. Should take the cube/s as input and return an instance of cf_units.Unit.

Returns An ifunc.

Example usage 1 Using an existing numpy ufunc, such as numpy.sin for the data function and a simple lambda function for the units function:

```
sine_ifunc = iris.analysis.maths.IFunc(
    numpy.sin, lambda cube: cf_units.Unit('1'))
sine_cube = sine_ifunc(cube)
```

Example usage 2 Define a function for the data arrays of two cubes and define a units function that checks the units of the cubes for consistency, before giving the resulting cube the same units as the first cube:

```
def ws_data_func(u_data, v_data):
    return numpy.sqrt( u_data**2 + v_data**2 )

def ws_units_func(u_cube, v_cube):
    if u_cube.units != getattr(v_cube, 'units', u_cube.
↪units):
        raise ValueError("units do not match")
    return u_cube.units

ws_ifunc = iris.analysis.maths.IFunc(ws_data_func, ws_units_
↪func)
ws_cube = ws_ifunc(u_cube, v_cube, new_name='wind speed')
```

Example usage 3 Using a data function that allows a keyword argument:

```
cs_ifunc = iris.analysis.maths.IFunc(numpy.cumsum,
    lambda a: a.units)
cs_cube = cs_ifunc(cube, axis=1)
```

__call__(*cube*, *other=None*, *dim=None*, *in_place=False*, *new_name=None*, ***kwargs_data_func*)

Applies the ifunc to the cube(s).

Args:

- **cube** An instance of *iris.cube.Cube*, whose data is used as the first argument to the data function.

Kwargs:

- **other** A cube, coord, ndarray or number whose data is used as the second argument to the data function.
- **new_name:** Name for the resulting Cube.
- **in_place:** Whether to create a new Cube, or alter the given "cube".
- **dim:** Dimension along which to apply *other* if it's a coordinate that is not found in *cube*
- **kwargs_data_func:** Keyword arguments that get passed on to the data_func.

Returns An instance of *iris.cube.Cube*.

27.1.5 iris.analysis.stats

Statistical operations between cubes.

In this module:

- *pearsonr*

iris.analysis.stats.**pearsonr**(*cube_a*, *cube_b*, *corr_coords=None*, *weights=None*, *mdtol=1.0*, *common_mask=False*)

Calculate the Pearson's r correlation coefficient over specified dimensions.

Args:

- **cube_a, cube_b (cubes):** Cubes between which the correlation will be calculated. The cubes should either be the same shape and have the same dimension coordinates or one cube should be broadcastable to the other.

- **corr_coords (str or list of str):** The cube coordinate name(s) over which to calculate correlations. If no names are provided then correlation will be calculated over all common cube dimensions.
- **weights (numpy.ndarray, optional):** Weights array of same shape as (the smaller of) cube_a and cube_b. Note that latitude/longitude area weights can be calculated using *iris.analysis.cartography.area_weights()*.
- **mdtol (float, optional):** Tolerance of missing data. The missing data fraction is calculated based on the number of grid cells masked in both cube_a and cube_b. If this fraction exceed mdtol, the returned value in the corresponding cell is masked. mdtol=0 means no missing data is tolerated while mdtol=1 means the resulting element will be masked if and only if all contributing elements are masked in cube_a or cube_b. Defaults to 1.
- **common_mask (bool):** If True, applies a common mask to cube_a and cube_b so only cells which are unmasked in both cubes contribute to the calculation. If False, the variance for each cube is calculated from all available cells. Defaults to False.

Returns

A cube of the correlation between the two input cubes along the specified dimensions, at each point in the remaining dimensions of the cubes.

For example providing two time/altitude/latitude/longitude cubes and corr_coords of 'latitude' and 'longitude' will result in a time/altitude cube describing the latitude/longitude (i.e. pattern) correlation at each time/altitude point.

Reference: https://en.wikipedia.org/wiki/Pearson_correlation_coefficient

This operation is non-lazy.

27.1.6 iris.analysis.trajectory

Defines a Trajectory class, and a routine to extract a sub-cube along a trajectory.

In this module:

- *interpolate*
- *Trajectory*
- *UnstructuredNearestNeigbourRegridder*

iris.analysis.trajectory.**interpolate**(*cube*, *sample_points*, *method=None*)

Extract a sub-cube at the given n-dimensional points.

Args:

- **cube** The source Cube.
- **sample_points** A sequence of coordinate (name) - values pairs.

Kwargs:

- **method** Request "linear" interpolation (default) or "nearest" neighbour. Only nearest neighbour is available when specifying multi-dimensional coordinates.

For example:

```
sample_points = [('latitude', [45, 45, 45]),
('longitude', [-60, -50, -40])]
interpolated_cube = interpolate(cube, sample_points)
```

A series of given waypoints with pre-calculated sample points.

class iris.analysis.trajectory.**Trajectory**(*waypoints*, *sample_count=10*)

Defines a trajectory using a sequence of waypoints.

For example:

```
waypoints = [{'latitude': 45, 'longitude': -60},
             {'latitude': 45, 'longitude': 0}]
Trajectory(waypoints)
```

Note: All the waypoint dictionaries must contain the same coordinate names.

Args:

- **waypoints** A sequence of dictionaries, mapping coordinate names to values.

Kwargs:

- **sample_count** The number of sample positions to use along the trajectory.

interpolate(*cube*, *method=None*)

Calls *interpolate()* to interpolate cube on the defined trajectory.

Assumes that the coordinate names supplied in the waypoints dictionaries match to coordinate names in *cube*, and that points are supplied in the same coord_system as in *cube*, where appropriate (i.e. for horizontal coordinate points).

Args:

- **cube** The source Cube to interpolate.

Kwargs:

- **method:** The interpolation method to use; "linear" (default) or "nearest". Only nearest is available when specifying multi-dimensional coordinates.

sampled_points

value}.

Type The trajectory points, as dictionaries of {coord_name

Encapsulate the operation of *iris.analysis.trajectory.interpolate()* with given source and target grids.

This is the type used by the *UnstructuredNearest* regridding scheme.

class iris.analysis.trajectory.**UnstructuredNearestNeigbourRegridder**(*src_cube*, *target_grid_cube*)

A nearest-neighbour regridder to perform regridding from the source grid to the target grid.

This can then be applied to any source data with the same structure as the original 'src_cube'.

Args:

- **src_cube:** The *Cube* defining the source grid. The X and Y coordinates can have any shape, but must be mapped over the same cube dimensions.
- **target_grid_cube:** A *Cube*, whose X and Y coordinates specify a desired target grid. The X and Y coordinates must be one-dimensional dimension coordinates, mapped to different dimensions. All other cube components are ignored.

Returns

(object)
A callable object with the interface: *result_cube = regridder(data)*
where *data* is a cube with the same grid as the original *src_cube*, that is to be regridded to the *target_grid_cube*.
Return type regridder

Note: For latitude-longitude coordinates, the nearest-neighbour distances are computed on the sphere, otherwise flat Euclidean distances are used.

The source and target X and Y coordinates must all have the same coordinate system, which may also be None. If any X and Y coordinates are latitudes or longitudes, they *all* must be. Otherwise, the corresponding X and Y coordinates must have the same units in the source and grid cubes.

A package providing *iris.cube.Cube* analysis support.

This module defines a suite of *Aggregator* instances, which are used to specify the statistical measure to calculate over a *Cube*, using methods such as *aggregated_by()* and *collapsed()*.

The *Aggregator* is a convenience class that allows specific statistical aggregation operators to be defined and instantiated. These operators can then be used to collapse, or partially collapse, one or more dimensions of a *Cube*, as discussed in *Cube Statistics*.

In particular, *Collapsing Entire Data Dimensions* discusses how to use *MEAN* to average over one dimension of a *Cube*, and also how to perform weighted *Area Averaging*. While *Partially Reducing Data Dimensions* shows how to aggregate similar groups of data points along a single dimension, to result in fewer points in that dimension.

The gallery contains several interesting worked examples of how an *Aggregator* may be used, including:

- *Global Average Annual Temperature Plot*
- *Applying a Filter to a Time-Series*
- *Hovmoller Diagram of Monthly Surface Temperature*
- *Seasonal Ensemble Model Plots*
- *Calculating a Custom Statistic*

In this module:

- *COUNT*
- *GMEAN*
- *HMEAN*

- *MAX*
- *MEAN*
- *MEDIAN*
- *MIN*
- *PEAK*
- *PERCENTILE*
- *PROPORTION*
- *RMS*
- *STD_DEV*
- *SUM*
- *VARIANCE*
- *WPERCENTILE*
- *Aggregator*
- *WeightedAggregator*
- *clear_phenomenon_identity*
- *Linear*
- *AreaWeighted*
- *Nearest*
- *UnstructuredNearest*
- *PointInCell*

iris.analysis.**COUNT** → Aggregator instance.

An *Aggregator* instance that counts the number of *Cube* data occurrences that satisfy a particular criterion, as defined by a user supplied *function*.

Required kwargs associated with the use of this aggregator:

- **function (callable):** A function which converts an array of data values into a corresponding array of True/False values.

For example:

To compute the number of *ensemble members* with precipitation exceeding 10 (in cube data units) could be calculated with:

```
result = precip_cube.collapsed('ensemble_member', iris.analysis.COUNT,
                               function=lambda values: values > 10)
```

See also:

The *PROPORTION()* aggregator.

This aggregator handles masked data.

iris.analysis.**GMEAN** → Aggregator instance.

An *Aggregator* instance that calculates the geometric mean over a *Cube*, as computed by scipy.stats.mstats.gmean().

For example:

To compute zonal geometric means over the *longitude* axis of a cube:

```
result = cube.collapsed('longitude', iris.analysis.GMEAN)
```

This aggregator handles masked data.

---

iris.analysis.**HMEAN** → Aggregator instance.

An *Aggregator* instance that calculates the harmonic mean over a *Cube*, as computed by scipy.stats.mstats.hmean().

For example:

To compute zonal harmonic mean over the *longitude* axis of a cube:

```
result = cube.collapsed('longitude', iris.analysis.HMEAN)
```

Note: The harmonic mean is only valid if all data values are greater than zero.

This aggregator handles masked data.

---

iris.analysis.**MAX** → Aggregator instance.

An *Aggregator* instance that calculates the maximum over a *Cube*, as computed by numpy.ma.max().

For example:

To compute zonal maximums over the *longitude* axis of a cube:

```
result = cube.collapsed('longitude', iris.analysis.MAX)
```

This aggregator handles masked data.

---

iris.analysis.**MEAN** → WeightedAggregator instance.

An *Aggregator* instance that calculates the mean over a *Cube*, as computed by numpy.ma.average().

Additional kwargs associated with the use of this aggregator:

- **mdtol (float):** Tolerance of missing data. The value returned in each element of the returned array will be masked if the fraction of masked data contributing to that element exceeds mdtol. This fraction is calculated based on the number of masked elements. mdtol=0 means no missing data is tolerated while mdtol=1 means the resulting element will be masked if and only if all the contributing elements are masked. Defaults to 1.
- **weights (float ndarray):** Weights matching the shape of the cube or the length of the window for rolling window operations. Note that, latitude/longitude area weights can be calculated using *iris.analysis.cartography.area_weights()*.

- **returned (boolean):** Set this to True to indicate that the collapsed weights are to be returned along with the collapsed data. Defaults to False.

For example:

To compute zonal means over the *longitude* axis of a cube:

```
result = cube.collapsed('longitude', iris.analysis.MEAN)
```

To compute a weighted area average:

```
coords = ('longitude', 'latitude')
collapsed_cube, collapsed_weights = cube.collapsed(coords,
                                                   iris.analysis.MEAN,
                                                   weights=weights,
                                                   returned=True)
```

Note: Lazy operation is supported, via `dask.array.ma.average()`.

This aggregator handles masked data.

`iris.analysis.`**`MEDIAN`** → Aggregator instance.

An *Aggregator* instance that calculates the median over a *Cube*, as computed by `numpy.ma.median()`.

For example:

To compute zonal medians over the *longitude* axis of a cube:

```
result = cube.collapsed('longitude', iris.analysis.MEDIAN)
```

This aggregator handles masked data.

`iris.analysis.`**`MIN`** → Aggregator instance.

An *Aggregator* instance that calculates the minimum over a *Cube*, as computed by `numpy.ma.min()`.

For example:

To compute zonal minimums over the *longitude* axis of a cube:

```
result = cube.collapsed('longitude', iris.analysis.MIN)
```

This aggregator handles masked data.

`iris.analysis.`**`PEAK`** → Aggregator instance.

An *Aggregator* instance that calculates the peak value derived from a spline interpolation over a *Cube*.

The peak calculation takes into account nan values. Therefore, if the number of non-nan values is zero the result itself will be an array of nan values.

The peak calculation also takes into account masked values. Therefore, if the number of non-masked values is zero the result itself will be a masked array.

If multiple coordinates are specified, then the peak calculations are performed individually, in sequence, for each coordinate specified.

For example:

To compute the peak over the *time* axis of a cube:

```
result = cube.collapsed('time', iris.analysis.PEAK)
```

This aggregator handles masked data.

---

iris.analysis.**PERCENTILE**

An PercentileAggregator instance that calculates the percentile over a *Cube*, as computed by scipy.stats.mstats.mquantiles().

Required kwargs associated with the use of this aggregator:

- **percent (float or sequence of floats):** Percentile rank/s at which to extract value/s.

Additional kwargs associated with the use of this aggregator:

- **alphap (float):** Plotting positions parameter, see scipy.stats.mstats.mquantiles(). Defaults to 1.
- **betap (float):** Plotting positions parameter, see scipy.stats.mstats.mquantiles(). Defaults to 1.

For example:

To compute the 10th and 90th percentile over *time*:

```
result = cube.collapsed('time', iris.analysis.PERCENTILE, percent=[10, 90])
```

This aggregator handles masked data.

---

iris.analysis.**PROPORTION** → Aggregator instance.

An *Aggregator* instance that calculates the proportion, as a fraction, of *Cube* data occurrences that satisfy a particular criterion, as defined by a user supplied *function*.

Required kwargs associated with the use of this aggregator:

- **function (callable):** A function which converts an array of data values into a corresponding array of True/False values.

For example:

To compute the probability of precipitation exceeding 10 (in cube data units) across *ensemble members* could be calculated with:

```
result = precip_cube.collapsed('ensemble_member', iris.analysis.PROPORTION,
                               function=lambda values: values > 10)
```

Similarly, the proportion of *time* precipitation exceeded 10 (in cube data units) could be calculated with:

```
result = precip_cube.collapsed('time', iris.analysis.PROPORTION,
                               function=lambda values: values > 10)
```

See also:

The *COUNT()* aggregator.

This aggregator handles masked data.

---

iris.analysis.**RMS** → WeightedAggregator instance.

An *Aggregator* instance that calculates the root mean square over a *Cube*, as computed by ((x0**2 + x1**2 + ... + xN-1**2) / N) ** 0.5.

Additional kwargs associated with the use of this aggregator:

- **weights (float ndarray):** Weights matching the shape of the cube or the length of the window for rolling window operations. The weights are applied to the squares when taking the mean.

For example:

To compute the zonal root mean square over the *longitude* axis of a cube:

```
result = cube.collapsed('longitude', iris.analysis.RMS)
```

This aggregator handles masked data.

---

iris.analysis.**STD_DEV** → Aggregator instance.

An *Aggregator* instance that calculates the standard deviation over a *Cube*, as computed by `numpy.ma.std()`.

Additional kwargs associated with the use of this aggregator:

- **ddof (integer):** Delta degrees of freedom. The divisor used in calculations is N - ddof, where N represents the number of elements. Defaults to 1.

For example:

To compute zonal standard deviations over the *longitude* axis of a cube:

```
result = cube.collapsed('longitude', iris.analysis.STD_DEV)
```

To obtain the biased standard deviation:

```
result = cube.collapsed('longitude', iris.analysis.STD_DEV, ddof=0)
```

Note: Lazy operation is supported, via `dask.array.nanstd()`.

This aggregator handles masked data.

---

iris.analysis.**SUM** → WeightedAggregator instance.

An *Aggregator* instance that calculates the sum over a *Cube*, as computed by `numpy.ma.sum()`.

Additional kwargs associated with the use of this aggregator:

- **weights (float ndarray):** Weights matching the shape of the cube, or the length of the window for rolling window operations. Weights should be normalized before using them with this aggregator if scaling is not intended.

- **returned (boolean):** Set this to True to indicate the collapsed weights are to be returned along with the collapsed data. Defaults to False.

For example:

To compute an accumulation over the *time* axis of a cube:

```
result = cube.collapsed('time', iris.analysis.SUM)
```

To compute a weighted rolling sum e.g. to apply a digital filter:

```
weights = np.array([.1, .2, .4, .2, .1])
result = cube.rolling_window('time', iris.analysis.SUM,
                             len(weights), weights=weights)
```

This aggregator handles masked data.

---

iris.analysis.**VARIANCE** → Aggregator instance.
An *Aggregator* instance that calculates the variance over a *Cube*, as computed by `numpy.ma.var()`.

Additional kwargs associated with the use of this aggregator:

- **ddof (integer):** Delta degrees of freedom. The divisor used in calculations is N - ddof, where N represents the number of elements. Defaults to 1.

For example:

To compute zonal variance over the *longitude* axis of a cube:

```
result = cube.collapsed('longitude', iris.analysis.VARIANCE)
```

To obtain the biased variance:

```
result = cube.collapsed('longitude', iris.analysis.VARIANCE, ddof=0)
```

Note: Lazy operation is supported, via `dask.array.nanvar()`.

This aggregator handles masked data.

---

iris.analysis.**WPERCENTILE**
An `WeightedPercentileAggregator` instance that calculates the weighted percentile over a *Cube*.

Required kwargs associated with the use of this aggregator:

- **percent (float or sequence of floats):** Percentile rank/s at which to extract value/s.
- **weights (float ndarray):** Weights matching the shape of the cube or the length of the window for rolling window operations. Note that, latitude/longitude area weights can be calculated using *iris.analysis.cartography.area_weights()*.

Additional kwargs associated with the use of this aggregator:

- **returned (boolean):** Set this to True to indicate that the collapsed weights are to be returned along with the collapsed data. Defaults to False.

- **kind (string or int):** Specifies the kind of interpolation used, see `scipy.interpolate.interp1d()` Defaults to "linear", which is equivalent to alphap=0.5, betap=0.5 in *iris.analysis.PERCENTILE*

The *Aggregator* class provides common aggregation functionality.

class iris.analysis.**Aggregator**(*cell_method*, *call_func*, *units_func=None*, *lazy_func=None*, ***kwargs*)

Create an aggregator for the given *call_func*.

Args:

- **cell_method (string):** Cell method definition formatter. Used in the fashion "cell_method.format(**kwargs)", to produce a cell-method string which can include keyword values.
- **call_func (callable):**

  Call signature: (data, axis=None, **kwargs)

  Data aggregation function. Returns an aggregation result, collapsing the 'axis' dimension of the 'data' argument.

Kwargs:

- **units_func (callable):**

  Call signature: (units)

  If provided, called to convert a cube's units. Returns an `cf_units.Unit`, or a value that can be made into one.
- **lazy_func (callable or None):** An alternative to *call_func* implementing a lazy aggregation. Note that, it need not support all features of the main operation, but should raise an error in unhandled cases.

Additional kwargs:: Passed through to *call_func* and *lazy_func*.

Aggregators are used by cube aggregation methods such as *collapsed()* and *aggregated_by()*. For example:

```
result = cube.collapsed('longitude', iris.analysis.MEAN)
```

A variety of ready-made aggregators are provided in this module, such as *MEAN* and *MAX*. Custom aggregators can also be created for special purposes, see *Calculating a Custom Statistic* for a worked example.

aggregate(*data*, *axis*, ***kwargs*)

Perform the aggregation function given the data.

Keyword arguments are passed through to the data aggregation function (for example, the "percent" keyword for a percentile aggregator). This function is usually used in conjunction with update_metadata(), which should be passed the same keyword arguments.

Args:

- **data (array):** Data array.
- **axis (int):** Axis to aggregate over.

Kwargs:

- **mdtol (float):** Tolerance of missing data. The value returned will be masked if the fraction of data to missing data is less than or equal to mdtol. mdtol=0 means no missing data is tolerated while mdtol=1 will return the resulting value from the aggregation function. Defaults to 1.
- **kwargs:** All keyword arguments apart from those specified above, are passed through to the data aggregation function.

Returns The aggregated data.

`aggregate_shape`(***kwargs*)
The shape of the new dimension/s created by the aggregator.

Kwargs:
- This function is intended to be used in conjunction with aggregate() and should be passed the same keywords.

Returns A tuple of the aggregate shape.

`lazy_aggregate`(*data*, *axis*, ***kwargs*)
Perform aggregation over the data with a lazy operation, analogous to the 'aggregate' result.

Keyword arguments are passed through to the data aggregation function (for example, the "percent" keyword for a percentile aggregator). This function is usually used in conjunction with update_metadata(), which should be passed the same keyword arguments.

Args:
- **data (array):** A lazy array (`dask.array.Array`).
- **axis (int or list of int):** The dimensions to aggregate over – note that this is defined differently to the 'aggregate' method 'axis' argument, which only accepts a single dimension index.

Kwargs:
- **kwargs:** All keyword arguments are passed through to the data aggregation function.

Returns A lazy array representing the aggregation operation (`dask.array.Array`).

`name`()
Returns the name of the aggregator.

`post_process`(*collapsed_cube*, *data_result*, *coords*, ***kwargs*)
Process the result from *iris.analysis.Aggregator.aggregate()*.

Args:
- **collapsed_cube:** A *iris.cube.Cube*.
- **data_result:** Result from *iris.analysis.Aggregator.aggregate()*
- **coords:** The one or more coordinates that were aggregated over.

Kwargs:
- This function is intended to be used in conjunction with aggregate() and should be passed the same keywords (for example, the "ddof" keyword from a standard deviation aggregator).

Returns The collapsed cube with its aggregated data payload.

`update_metadata`(*cube*, *coords*, ***kwargs*)
Update cube cell method metadata w.r.t the aggregation function.

Args:
- **cube (*iris.cube.Cube*):** Source cube that requires metadata update.

- **coords (*iris.coords.Coord*):** The one or more coordinates that were aggregated.

Kwargs:

- This function is intended to be used in conjunction with aggregate() and should be passed the same keywords (for example, the "ddof" keyword for a standard deviation aggregator).

call_func
Data aggregation function.

cell_method
Cube cell method string.

lazy_func
Lazy aggregation function, may be None to indicate that a lazy operation is not available.

units_func
Unit conversion function.

Convenience class that supports common weighted aggregation functionality.

class iris.analysis.**WeightedAggregator**(*cell_method*, *call_func*, *units_func=None*, *lazy_func=None*, ***kwargs*)

Create a weighted aggregator for the given *call_func*.

Args:

- **cell_method (string):** Cell method string that supports string format substitution.
- **call_func (callable):** Data aggregation function. Call signature *(data, axis, **kwargs)*.

Kwargs:

- **units_func (callable):** Units conversion function.
- **lazy_func (callable or None):** An alternative to *call_func* implementing a lazy aggregation. Note that, it need not support all features of the main operation, but should raise an error in unhandled cases.

Additional kwargs: Passed through to *call_func* and *lazy_func*.

aggregate(*data*, *axis*, ***kwargs*)
Perform the aggregation function given the data.

Keyword arguments are passed through to the data aggregation function (for example, the "percent" keyword for a percentile aggregator). This function is usually used in conjunction with update_metadata(), which should be passed the same keyword arguments.

Args:

- **data (array):** Data array.
- **axis (int):** Axis to aggregate over.

Kwargs:

- **mdtol (float):** Tolerance of missing data. The value returned will be masked if the fraction of data to missing data is less than or equal to mdtol. mdtol=0 means no missing data is tolerated while mdtol=1 will return the resulting value from the aggregation function. Defaults to 1.

- **kwargs:** All keyword arguments apart from those specified above, are passed through to the data aggregation function.

Returns The aggregated data.

aggregate_shape(***kwargs*)
The shape of the new dimension/s created by the aggregator.

Kwargs:
- This function is intended to be used in conjunction with aggregate() and should be passed the same keywords.

Returns A tuple of the aggregate shape.

lazy_aggregate(*data*, *axis*, ***kwargs*)
Perform aggregation over the data with a lazy operation, analogous to the 'aggregate' result.

Keyword arguments are passed through to the data aggregation function (for example, the "percent" keyword for a percentile aggregator). This function is usually used in conjunction with update_metadata(), which should be passed the same keyword arguments.

Args:
- **data (array):** A lazy array (`dask.array.Array`).
- **axis (int or list of int):** The dimensions to aggregate over – note that this is defined differently to the 'aggregate' method 'axis' argument, which only accepts a single dimension index.

Kwargs:
- **kwargs:** All keyword arguments are passed through to the data aggregation function.

Returns A lazy array representing the aggregation operation (`dask.array.Array`).

name()
Returns the name of the aggregator.

post_process(*collapsed_cube*, *data_result*, *coords*, ***kwargs*)
Process the result from *iris.analysis.Aggregator.aggregate()*.

Returns a tuple(cube, weights) if a tuple(data, weights) was returned from *iris.analysis.Aggregator.aggregate()*.

Args:
- **collapsed_cube:** A *iris.cube.Cube*.
- **data_result:** Result from *iris.analysis.Aggregator.aggregate()*
- **coords:** The one or more coordinates that were aggregated over.

Kwargs:
- This function is intended to be used in conjunction with aggregate() and should be passed the same keywords (for example, the "weights" keywords from a mean aggregator).

Returns The collapsed cube with it's aggregated data payload. Or a tuple pair of (cube, weights) if the keyword "returned" is specified and True.

update_metadata(*cube*, *coords*, ***kwargs*)
Update cube cell method metadata w.r.t the aggregation function.

Args:
- **cube (*iris.cube.Cube*):** Source cube that requires metadata update.

- **coords (*iris.coords.Coord*):** The one or more coordinates that were aggregated.

Kwargs:

- This function is intended to be used in conjunction with aggregate() and should be passed the same keywords (for example, the "ddof" keyword for a standard deviation aggregator).

uses_weighting(***kwargs*)
Determine whether this aggregator uses weighting.

Kwargs:

- **kwargs:** Arguments to filter of weighted keywords.

Returns Boolean.

call_func
Data aggregation function.

cell_method
Cube cell method string.

lazy_func
Lazy aggregation function, may be None to indicate that a lazy operation is not available.

units_func
Unit conversion function.

---

iris.analysis.**clear_phenomenon_identity**(*cube*)
Helper function to clear the standard_name, attributes, and cell_methods of a cube.

This class describes the linear interpolation and regridding scheme for interpolating or regridding over one or more orthogonal coordinates, typically for use with *iris.cube.Cube.interpolate()* or *iris.cube.Cube.regrid()*.

class iris.analysis.**Linear**(*extrapolation_mode='linear'*)
Linear interpolation and regridding scheme suitable for interpolating or regridding over one or more orthogonal coordinates.

Kwargs:

- **extrapolation_mode:** Must be one of the following strings:
  - 'extrapolate' or 'linear' - The extrapolation points will be calculated by extending the gradient of the closest two points.
  - 'nan' - The extrapolation points will be be set to NaN.
  - 'error' - A ValueError exception will be raised, notifying an attempt to extrapolate.
  - 'mask' - The extrapolation points will always be masked, even if the source data is not a MaskedArray.
  - 'nanmask' - If the source data is a MaskedArray the extrapolation points will be masked. Otherwise they will be set to NaN.

  The default mode of extrapolation is 'linear'.

interpolator(*cube*, *coords*)

Creates a linear interpolator to perform interpolation over the given *Cube* specified by the dimensions of the given coordinates.

Typically you should use *iris.cube.Cube.interpolate()* for interpolating a cube. There are, however, some situations when constructing your own interpolator is preferable. These are detailed in the *user guide*.

Args:

- **cube:** The source *iris.cube.Cube* to be interpolated.
- **coords:** The names or coordinate instances that are to be interpolated over.

Returns

callable(sample_points, collapse_scalar=True)

where *sample_points* is a sequence containing an array of values for each of the coordinates passed to this method, and *collapse_scalar* determines whether to remove length one dimensions in the result cube caused by scalar values in *sample_points*.

The values for coordinates that correspond to date/times may optionally be supplied as datetime.datetime or cftime.datetime instances.

For example, for the callable returned by: *Linear().interpolator(cube, ['latitude', 'longitude'])*, sample_points must have the form *[new_lat_values, new_lon_values]*.

Return type A callable with the interface

regridder(*src_grid*, *target_grid*)

Creates a linear regridder to perform regridding from the source grid to the target grid.

Typically you should use *iris.cube.Cube.regrid()* for regridding a cube. There are, however, some situations when constructing your own regridder is preferable. These are detailed in the *user guide*.

Supports lazy regridding. Any chunks in horizontal dimensions will be combined before regridding.

Args:

- **src_grid:** The *Cube* defining the source grid.
- **target_grid:** The *Cube* defining the target grid.

Returns

callable(cube)

where *cube* is a cube with the same grid as *src_grid* that is to be regridded to the *target_grid*.

Return type A callable with the interface

LINEAR_EXTRAPOLATION_MODES = ['extrapolate', 'error', 'nan', 'mask', 'nanmask', 'linear']

This class describes an area-weighted regridding scheme for regridding between 'ordinary' horizontal grids with separated X and Y coordinates in a common coordinate system. Typically for use with *iris.cube.Cube.regrid()*.

class iris.analysis.**AreaWeighted**(*mdtol=1*)

Area-weighted regridding scheme suitable for regridding between different orthogonal XY grids in the same coordinate system.

Kwargs:

- **mdtol (float):** Tolerance of missing data. The value returned in each element of the returned array will be masked if the fraction of missing data exceeds mdtol. This fraction is calculated based on the area of masked cells within each target cell. mdtol=0 means no masked data is tolerated while mdtol=1 will mean the resulting element will be masked if and only if all the overlapping elements of the source grid are masked. Defaults to 1.

regridder(*src_grid_cube*, *target_grid_cube*)

Creates an area-weighted regridder to perform regridding from the source grid to the target grid.

Typically you should use *iris.cube.Cube.regrid()* for regridding a cube. There are, however, some situations when constructing your own regridder is preferable. These are detailed in the *user guide*.

Supports lazy regridding. Any chunks in horizontal dimensions will be combined before regridding.

Args:

- **src_grid_cube:** The *Cube* defining the source grid.
- **target_grid_cube:** The *Cube* defining the target grid.

Returns

callable(cube)

where *cube* is a cube with the same grid as *src_grid_cube* that is to be regridded to the grid of *target_grid_cube*.

Return type A callable with the interface

This class describes the nearest-neighbour interpolation and regridding scheme for interpolating or regridding over one or more orthogonal coordinates, typically for use with *iris.cube.Cube.interpolate()* or *iris.cube.Cube.regrid()*.

class iris.analysis.**Nearest**(*extrapolation_mode='extrapolate'*)

Nearest-neighbour interpolation and regridding scheme suitable for interpolating or regridding over one or more orthogonal coordinates.

Kwargs:

- **extrapolation_mode:** Must be one of the following strings:
  - 'extrapolate' - The extrapolation points will take their value from the nearest source point.
  - 'nan' - The extrapolation points will be be set to NaN.
  - 'error' - A ValueError exception will be raised, notifying an attempt to extrapolate.
  - 'mask' - The extrapolation points will always be masked, even if the source data is not a MaskedArray.
  - 'nanmask' - If the source data is a MaskedArray the extrapolation points will be masked. Otherwise they will be set to NaN.

The default mode of extrapolation is 'extrapolate'.

interpolator(*cube*, *coords*)

Creates a nearest-neighbour interpolator to perform interpolation over the given *Cube* specified by the dimensions of the specified coordinates.

Typically you should use *iris.cube.Cube.interpolate()* for interpolating a cube. There are, however, some situations when constructing your own interpolator is preferable. These are detailed in the *user guide*.

Args:

- **cube:** The source *iris.cube.Cube* to be interpolated.
- **coords:** The names or coordinate instances that are to be interpolated over.

Returns

callable(sample_points, collapse_scalar=True)

where *sample_points* is a sequence containing an array of values for each of the coordinates passed to this method, and *collapse_scalar* determines whether to remove length one dimensions in the result cube caused by scalar values in *sample_points*.

The values for coordinates that correspond to date/times may optionally be supplied as datetime.datetime or cftime.datetime instances.

For example, for the callable returned by: *Nearest().interpolator(cube, ['latitude', 'longitude'])*, sample_points must have the form *[new_lat_values, new_lon_values]*.

Return type A callable with the interface

regridder(*src_grid*, *target_grid*)

Creates a nearest-neighbour regridder to perform regridding from the source grid to the target grid.

Typically you should use *iris.cube.Cube.regrid()* for regridding a cube. There are, however, some situations when constructing your own regridder is preferable. These are detailed in the *user guide*.

Supports lazy regridding. Any chunks in horizontal dimensions will be combined before regridding.

Args:

- **src_grid:** The *Cube* defining the source grid.
- **target_grid:** The *Cube* defining the target grid.

Returns

callable(cube)

where *cube* is a cube with the same grid as *src_grid* that is to be regridded to the *target_grid*.

Return type A callable with the interface

This is a nearest-neighbour regridding scheme for regridding data whose horizontal (X- and Y-axis) coordinates are mapped to the *same* dimensions, rather than being orthogonal on independent dimensions.

For latitude-longitude coordinates, the nearest-neighbour distances are computed on the sphere, otherwise flat Euclidean distances are used.

The source X and Y coordinates can have any shape.

The target grid must be of the "normal" kind, i.e. it has separate, 1-dimensional X and Y coordinates.

Source and target XY coordinates must have the same coordinate system, which may also be None. If any of the XY coordinates are latitudes or longitudes, then they *all* must be. Otherwise, the corresponding X and Y coordinates must have the same units in the source and grid cubes.

Note: Currently only supports regridding, not interpolation.

Note: This scheme performs essentially the same job as *iris.experimental.regrid.ProjectedUnstructuredNearest*. That scheme is faster, but only works well on data in a limited region of the globe, covered by a specified projection. This approach is more rigorously correct and can be applied to global datasets.

---

class iris.analysis.**UnstructuredNearest**

Nearest-neighbour interpolation and regridding scheme suitable for interpolating or regridding from un-gridded data such as trajectories or other data where the X and Y coordinates share the same dimensions.

regridder(*src_cube*, *target_grid*)

Creates a nearest-neighbour regridder, of the *UnstructuredNearestNeigbourRegridder* type, to perform regridding from the source grid to the target grid.

This can then be applied to any source data with the same structure as the original 'src_cube'.

Typically you should use *iris.cube.Cube.regrid()* for regridding a cube. There are, however, some situations when constructing your own regridder is preferable. These are detailed in the *user guide*.

Does not support lazy regridding.

Args:

- **src_cube:** The *Cube* defining the source grid. The X and Y coordinates can have any shape, but must be mapped over the same cube dimensions.
- **target_grid:** The *Cube* defining the target grid. The X and Y coordinates must be one-dimensional dimension coordinates, mapped to different dimensions. All other cube components are ignored.

Returns

callable(cube)

where *cube* is a cube with the same grid as *src_cube* that is to be regridded to the *target_grid*.

Return type A callable with the interface

This class describes the point-in-cell regridding scheme for use typically with *iris.cube.Cube.regrid()*.

The PointInCell regridder can regrid data from a source grid of any dimensionality and in any coordinate system. The location of each source point is specified by X and Y coordinates mapped over the same cube dimensions, aka "grid dimensions" : the grid may have any dimensionality. The X and Y coordinates must also have the same, defined coord_system. The weights, if specified, must have the same shape as the X and Y coordinates. The output grid can be any 'normal' XY grid, specified by *separate* X and Y

coordinates : That is, X and Y have two different cube dimensions. The output X and Y coordinates must also have a common, specified coord_system.

class iris.analysis.**PointInCell**(*weights=None*)

Point-in-cell regridding scheme suitable for regridding over one or more orthogonal coordinates.

Optional Args:

- **weights:** A numpy.ndarray instance that defines the weights for the grid cells of the source grid. Must have the same shape as the data of the source grid. If unspecified, equal weighting is assumed.

regridder(*src_grid*, *target_grid*)

Creates a point-in-cell regridder to perform regridding from the source grid to the target grid.

Typically you should use *iris.cube.Cube.regrid()* for regridding a cube. There are, however, some situations when constructing your own regridder is preferable. These are detailed in the *user guide*.

Does not support lazy regridding.

Args:

- **src_grid:** The *Cube* defining the source grid.
- **target_grid:** The *Cube* defining the target grid.

Returns

callable(cube)

where *cube* is a cube with the same grid as *src_grid* that is to be regridded to the *target_grid*.

Return type A callable with the interface

27.2 iris.aux_factory

Definitions of derived coordinates.

In this module:

- *AtmosphereSigmaFactory*
- *AuxCoordFactory*
- *HybridHeightFactory*
- *HybridPressureFactory*
- *OceanSFactory*
- *OceanSg1Factory*
- *OceanSg2Factory*
- *OceanSigmaFactory*
- *OceanSigmaZFactory*

Defines an atmosphere sigma coordinate factory with the formula: p = ptop + sigma * (ps - ptop)

class iris.aux_factory.**AtmosphereSigmaFactory**(*pressure_at_top=None*, *sigma=None*, *surface_air_pressure=None*)

Creates an atmosphere sigma coordinate factory with the formula:

p(n, k, j, i) = pressure_at_top + sigma(k) * (surface_air_pressure(n, j, i) - pressure_at_top)

derived_dims(*coord_dims_func*)

Returns the cube dimensions for the derived coordinate.

Args:

- **coord_dims_func:** A callable which can return the list of dimensions relevant to a given coordinate. See *iris.cube.Cube.coord_dims()*.

Returns A sorted list of cube dimension numbers.

make_coord(*coord_dims_func*)

Returns a new *iris.coords.AuxCoord* as defined by this factory.

Args:

- **coord_dims_func:** A callable which can return the list of dimensions relevant to a given coordinate. See *iris.cube.Cube.coord_dims()*.

name(*default=None*, *token=False*)

Returns a string name representing the identity of the metadata.

First it tries standard name, then it tries the long name, then the NetCDF variable name, before falling-back to a default value, which itself defaults to the string 'unknown'.

Kwargs:

- **default:** The fall-back string representing the default name. Defaults to the string 'unknown'.
- **token:** If True, ensures that the name returned satisfies the criteria for the characters required by a valid NetCDF name. If it is not possible to return a valid name, then a ValueError exception is raised. Defaults to False.

Returns String.

rename(*name*)

Changes the human-readable name.

If 'name' is a valid standard name it will assign it to *standard_name*, otherwise it will assign it to *long_name*.

update(*old_coord*, *new_coord=None*)

Notifies the factory of the removal/replacement of a coordinate which might be a dependency.

Args:

- **old_coord:** The coordinate to be removed/replaced.
- **new_coord:** If None, any dependency using old_coord is removed, otherwise any dependency using old_coord is updated to use new_coord.

updated(*new_coord_mapping*)

Creates a new instance of this factory where the dependencies are replaced according to the given mapping.

Args:

- **new_coord_mapping:** A dictionary mapping from the object IDs potentially used by this factory, to the coordinate objects that should be used instead.

xml_element(*doc*)
Returns a DOM element describing this coordinate factory.

property attributes

property climatological
Always returns False, as a factory itself can never have points/bounds and therefore can never be climatological by definition.

property coord_system
The coordinate-system (if any) of the coordinate made by the factory.

property dependencies
Return dependencies.

property long_name
The CF Metadata long name for the object.

property metadata

property standard_name
The CF Metadata standard name for the object.

property units
The S.I. unit of the object.

property var_name
The NetCDF variable name for the object.

Represents a "factory" which can manufacture an additional auxiliary coordinate on demand, by combining the values of other coordinates.

Each concrete subclass represents a specific formula for deriving values from other coordinates.

The *standard_name*, *long_name*, *var_name*, *units*, *attributes* and *coord_system* of the factory are used to set the corresponding properties of the resulting auxiliary coordinates.

class iris.aux_factory.**AuxCoordFactory**
Represents a "factory" which can manufacture an additional auxiliary coordinate on demand, by combining the values of other coordinates.

Each concrete subclass represents a specific formula for deriving values from other coordinates.

The *standard_name*, *long_name*, *var_name*, *units*, *attributes* and *coord_system* of the factory are used to set the corresponding properties of the resulting auxiliary coordinates.

derived_dims(*coord_dims_func*)
Returns the cube dimensions for the derived coordinate.

Args:
- **coord_dims_func:** A callable which can return the list of dimensions relevant to a given coordinate. See *iris.cube.Cube.coord_dims()*.

Returns A sorted list of cube dimension numbers.

abstract make_coord(*coord_dims_func*)
Returns a new *iris.coords.AuxCoord* as defined by this factory.

Args:

- **coord_dims_func:** A callable which can return the list of dimensions relevant to a given coordinate. See *iris.cube.Cube.coord_dims()*.

name(*default=None*, *token=False*)
Returns a string name representing the identity of the metadata.

First it tries standard name, then it tries the long name, then the NetCDF variable name, before falling-back to a default value, which itself defaults to the string 'unknown'.

Kwargs:
- **default:** The fall-back string representing the default name. Defaults to the string 'unknown'.
- **token:** If True, ensures that the name returned satisfies the criteria for the characters required by a valid NetCDF name. If it is not possible to return a valid name, then a ValueError exception is raised. Defaults to False.

Returns String.

rename(*name*)
Changes the human-readable name.

If 'name' is a valid standard name it will assign it to *standard_name*, otherwise it will assign it to *long_name*.

update(*old_coord*, *new_coord=None*)
Notifies the factory of the removal/replacement of a coordinate which might be a dependency.

Args:
- **old_coord:** The coordinate to be removed/replaced.
- **new_coord:** If None, any dependency using old_coord is removed, otherwise any dependency using old_coord is updated to use new_coord.

updated(*new_coord_mapping*)
Creates a new instance of this factory where the dependencies are replaced according to the given mapping.

Args:
- **new_coord_mapping:** A dictionary mapping from the object IDs potentially used by this factory, to the coordinate objects that should be used instead.

xml_element(*doc*)
Returns a DOM element describing this coordinate factory.

property attributes

property climatological
Always returns False, as a factory itself can never have points/bounds and therefore can never be climatological by definition.

property coord_system
The coordinate-system (if any) of the coordinate made by the factory.

abstract property dependencies
Returns a dictionary mapping from constructor argument names to the corresponding coordinates.

property long_name
Descriptive name of the coordinate made by the factory

property metadata

property standard_name
The CF Metadata standard name for the object.

property units
The S.I. unit of the object.

property var_name
netCDF variable name for the coordinate made by the factory

Defines a hybrid-height coordinate factory with the formula: z = a + b * orog

class iris.aux_factory.**HybridHeightFactory**(*delta=None*, *sigma=None*, *orography=None*)

Creates a hybrid-height coordinate factory with the formula: z = a + b * orog

At least one of *delta* or *orography* must be provided.

Args:

- **delta: Coord** The coordinate providing the *a* term.
- **sigma: Coord** The coordinate providing the *b* term.
- **orography: Coord** The coordinate providing the *orog* term.

derived_dims(*coord_dims_func*)
Returns the cube dimensions for the derived coordinate.

Args:
- **coord_dims_func:** A callable which can return the list of dimensions relevant to a given coordinate. See *iris.cube.Cube.coord_dims()*.

Returns A sorted list of cube dimension numbers.

make_coord(*coord_dims_func*)
Returns a new *iris.coords.AuxCoord* as defined by this factory.

Args:
- **coord_dims_func:** A callable which can return the list of dimensions relevant to a given coordinate. See *iris.cube.Cube.coord_dims()*.

name(*default=None*, *token=False*)
Returns a string name representing the identity of the metadata.

First it tries standard name, then it tries the long name, then the NetCDF variable name, before falling-back to a default value, which itself defaults to the string 'unknown'.

Kwargs:
- **default:** The fall-back string representing the default name. Defaults to the string 'unknown'.
- **token:** If True, ensures that the name returned satisfies the criteria for the characters required by a valid NetCDF name. If it is not possible to return a valid name, then a ValueError exception is raised. Defaults to False.

Returns String.

rename(*name*)
Changes the human-readable name.

If 'name' is a valid standard name it will assign it to *standard_name*, otherwise it will assign it to *long_name*.

update(*old_coord*, *new_coord=None*)
Notifies the factory of the removal/replacement of a coordinate which might be a dependency.

Args:
- **old_coord:** The coordinate to be removed/replaced.
- **new_coord:** If None, any dependency using old_coord is removed, otherwise any dependency using old_coord is updated to use new_coord.

updated(*new_coord_mapping*)
Creates a new instance of this factory where the dependencies are replaced according to the given mapping.

Args:
- **new_coord_mapping:** A dictionary mapping from the object IDs potentially used by this factory, to the coordinate objects that should be used instead.

xml_element(*doc*)
Returns a DOM element describing this coordinate factory.

property attributes

property climatological
Always returns False, as a factory itself can never have points/bounds and therefore can never be climatological by definition.

property coord_system
The coordinate-system (if any) of the coordinate made by the factory.

property dependencies
Returns a dictionary mapping from constructor argument names to the corresponding coordinates.

property long_name
The CF Metadata long name for the object.

property metadata

property standard_name
The CF Metadata standard name for the object.

property units
The S.I. unit of the object.

property var_name
The NetCDF variable name for the object.

Defines a hybrid-pressure coordinate factory with the formula: p = ap + b * ps

class iris.aux_factory.**HybridPressureFactory**(*delta=None*, *sigma=None*, *surface_air_pressure=None*)

Creates a hybrid-height coordinate factory with the formula: p = ap + b * ps

At least one of *delta* or *surface_air_pressure* must be provided.

Args:

- **delta: Coord** The coordinate providing the *ap* term.
- **sigma: Coord** The coordinate providing the *b* term.
- **surface_air_pressure: Coord** The coordinate providing the *ps* term.

derived_dims(*coord_dims_func*)
Returns the cube dimensions for the derived coordinate.

Args:
- **coord_dims_func:** A callable which can return the list of dimensions relevant to a given coordinate. See *iris.cube.Cube.coord_dims()*.

Returns A sorted list of cube dimension numbers.

make_coord(*coord_dims_func*)
Returns a new *iris.coords.AuxCoord* as defined by this factory.

Args:
- **coord_dims_func:** A callable which can return the list of dimensions relevant to a given coordinate. See *iris.cube.Cube.coord_dims()*.

name(*default=None*, *token=False*)
Returns a string name representing the identity of the metadata.

First it tries standard name, then it tries the long name, then the NetCDF variable name, before falling-back to a default value, which itself defaults to the string 'unknown'.

Kwargs:
- **default:** The fall-back string representing the default name. Defaults to the string 'unknown'.
- **token:** If True, ensures that the name returned satisfies the criteria for the characters required by a valid NetCDF name. If it is not possible to return a valid name, then a ValueError exception is raised. Defaults to False.

Returns String.

rename(*name*)
Changes the human-readable name.

If 'name' is a valid standard name it will assign it to *standard_name*, otherwise it will assign it to *long_name*.

update(*old_coord*, *new_coord=None*)
Notifies the factory of the removal/replacement of a coordinate which might be a dependency.

Args:
- **old_coord:** The coordinate to be removed/replaced.
- **new_coord:** If None, any dependency using old_coord is removed, otherwise any dependency using old_coord is updated to use new_coord.

updated(*new_coord_mapping*)
Creates a new instance of this factory where the dependencies are replaced according to the given mapping.

Args:
- **new_coord_mapping:** A dictionary mapping from the object IDs potentially used by this factory, to the coordinate objects that should be used instead.

xml_element(*doc*)
Returns a DOM element describing this coordinate factory.

property attributes

property climatological
Always returns False, as a factory itself can never have points/bounds and therefore can never be climatological by definition.

property coord_system
The coordinate-system (if any) of the coordinate made by the factory.

property dependencies
Returns a dictionary mapping from constructor argument names to the corresponding coordinates.

property long_name
The CF Metadata long name for the object.

property metadata

property standard_name
The CF Metadata standard name for the object.

property units
The S.I. unit of the object.

property var_name
The NetCDF variable name for the object.

Defines an Ocean s-coordinate factory.

class iris.aux_factory.**OceanSFactory**(*s=None*, *eta=None*, *depth=None*, *a=None*, *b=None*, *depth_c=None*)
Creates an Ocean s-coordinate factory with the formula:

z(n,k,j,i) = eta(n,j,j,i)*(1+s(k)) + depth_c*s(k) + (depth(j,i)-depth_c)*C(k)

where:

C(k) = (1-b) * sinh(a*s(k)) / sinh(a) + b * [tanh(a * (s(k) + 0.5)) / (2 * tanh(0.5*a)) - 0.5]

derived_dims(*coord_dims_func*)
Returns the cube dimensions for the derived coordinate.

Args:
- **coord_dims_func:** A callable which can return the list of dimensions relevant to a given coordinate. See *iris.cube.Cube.coord_dims()*.

Returns A sorted list of cube dimension numbers.

make_coord(*coord_dims_func*)
Returns a new *iris.coords.AuxCoord* as defined by this factory.

Args:
- **coord_dims_func:** A callable which can return the list of dimensions relevant to a given coordinate. See *iris.cube.Cube.coord_dims()*.

name(*default=None*, *token=False*)
Returns a string name representing the identity of the metadata.

First it tries standard name, then it tries the long name, then the NetCDF variable name, before falling-back to a default value, which itself defaults to the string 'unknown'.

Kwargs:

- **default:** The fall-back string representing the default name. Defaults to the string 'unknown'.
- **token:** If True, ensures that the name returned satisfies the criteria for the characters required by a valid NetCDF name. If it is not possible to return a valid name, then a ValueError exception is raised. Defaults to False.

Returns String.

rename(*name*)

Changes the human-readable name.

If 'name' is a valid standard name it will assign it to *standard_name*, otherwise it will assign it to *long_name*.

update(*old_coord*, *new_coord=None*)

Notifies the factory of the removal/replacement of a coordinate which might be a dependency.

Args:

- **old_coord:** The coordinate to be removed/replaced.
- **new_coord:** If None, any dependency using old_coord is removed, otherwise any dependency using old_coord is updated to use new_coord.

updated(*new_coord_mapping*)

Creates a new instance of this factory where the dependencies are replaced according to the given mapping.

Args:

- **new_coord_mapping:** A dictionary mapping from the object IDs potentially used by this factory, to the coordinate objects that should be used instead.

xml_element(*doc*)

Returns a DOM element describing this coordinate factory.

property attributes

property climatological

Always returns False, as a factory itself can never have points/bounds and therefore can never be climatological by definition.

property coord_system

The coordinate-system (if any) of the coordinate made by the factory.

property dependencies

Returns a dictionary mapping from constructor argument names to the corresponding coordinates.

property long_name

The CF Metadata long name for the object.

property metadata

property standard_name

The CF Metadata standard name for the object.

property units

The S.I. unit of the object.

property var_name

The NetCDF variable name for the object.

Defines an Ocean s-coordinate, generic form 1 factory.

class iris.aux_factory.**OceanSg1Factory**(*s=None*, *c=None*, *eta=None*, *depth=None*, *depth_c=None*)

Creates an Ocean s-coordinate, generic form 1 factory with the formula:

z(n,k,j,i) = S(k,j,i) + eta(n,j,i) * (1 + S(k,j,i) / depth(j,i))

where: S(k,j,i) = depth_c * s(k) + (depth(j,i) - depth_c) * C(k)

derived_dims(*coord_dims_func*)

Returns the cube dimensions for the derived coordinate.

Args:

- **coord_dims_func:** A callable which can return the list of dimensions relevant to a given coordinate. See *iris.cube.Cube.coord_dims()*.

Returns A sorted list of cube dimension numbers.

make_coord(*coord_dims_func*)

Returns a new *iris.coords.AuxCoord* as defined by this factory.

Args:

- **coord_dims_func:** A callable which can return the list of dimensions relevant to a given coordinate. See *iris.cube.Cube.coord_dims()*.

name(*default=None*, *token=False*)

Returns a string name representing the identity of the metadata.

First it tries standard name, then it tries the long name, then the NetCDF variable name, before falling-back to a default value, which itself defaults to the string 'unknown'.

Kwargs:

- **default:** The fall-back string representing the default name. Defaults to the string 'unknown'.
- **token:** If True, ensures that the name returned satisfies the criteria for the characters required by a valid NetCDF name. If it is not possible to return a valid name, then a ValueError exception is raised. Defaults to False.

Returns String.

rename(*name*)

Changes the human-readable name.

If 'name' is a valid standard name it will assign it to *standard_name*, otherwise it will assign it to *long_name*.

update(*old_coord*, *new_coord=None*)

Notifies the factory of the removal/replacement of a coordinate which might be a dependency.

Args:

- **old_coord:** The coordinate to be removed/replaced.
- **new_coord:** If None, any dependency using old_coord is removed, otherwise any dependency using old_coord is updated to use new_coord.

updated(*new_coord_mapping*)

Creates a new instance of this factory where the dependencies are replaced according to the given mapping.

Args:

- **new_coord_mapping:** A dictionary mapping from the object IDs potentially used by this factory, to the coordinate objects that should be used instead.

xml_element(*doc*)
Returns a DOM element describing this coordinate factory.

property attributes

property climatological
Always returns False, as a factory itself can never have points/bounds and therefore can never be climatological by definition.

property coord_system
The coordinate-system (if any) of the coordinate made by the factory.

property dependencies
Returns a dictionary mapping from constructor argument names to the corresponding coordinates.

property long_name
The CF Metadata long name for the object.

property metadata

property standard_name
The CF Metadata standard name for the object.

property units
The S.I. unit of the object.

property var_name
The NetCDF variable name for the object.

Defines an Ocean s-coordinate, generic form 2 factory.

class iris.aux_factory.**OceanSg2Factory**(*s=None*, *c=None*, *eta=None*, *depth=None*, *depth_c=None*)
Creates an Ocean s-coordinate, generic form 2 factory with the formula:

z(n,k,j,i) = eta(n,j,i) + (eta(n,j,i) + depth(j,i)) * S(k,j,i)

where:

S(k,j,i) = (depth_c * s(k) + depth(j,i) * C(k)) / (depth_c + depth(j,i))

derived_dims(*coord_dims_func*)
Returns the cube dimensions for the derived coordinate.

Args:

- **coord_dims_func:** A callable which can return the list of dimensions relevant to a given coordinate. See *iris.cube.Cube.coord_dims()*.

Returns A sorted list of cube dimension numbers.

make_coord(*coord_dims_func*)
Returns a new *iris.coords.AuxCoord* as defined by this factory.

Args:

- **coord_dims_func:** A callable which can return the list of dimensions relevant to a given coordinate. See *iris.cube.Cube.coord_dims()*.

name(*default=None*, *token=False*)
Returns a string name representing the identity of the metadata.

First it tries standard name, then it tries the long name, then the NetCDF variable name, before falling-back to a default value, which itself defaults to the string 'unknown'.

Kwargs:
- **default:** The fall-back string representing the default name. Defaults to the string 'unknown'.
- **token:** If True, ensures that the name returned satisfies the criteria for the characters required by a valid NetCDF name. If it is not possible to return a valid name, then a ValueError exception is raised. Defaults to False.

Returns String.

rename(*name*)
Changes the human-readable name.

If 'name' is a valid standard name it will assign it to *standard_name*, otherwise it will assign it to *long_name*.

update(*old_coord*, *new_coord=None*)
Notifies the factory of the removal/replacement of a coordinate which might be a dependency.

Args:
- **old_coord:** The coordinate to be removed/replaced.
- **new_coord:** If None, any dependency using old_coord is removed, otherwise any dependency using old_coord is updated to use new_coord.

updated(*new_coord_mapping*)
Creates a new instance of this factory where the dependencies are replaced according to the given mapping.

Args:
- **new_coord_mapping:** A dictionary mapping from the object IDs potentially used by this factory, to the coordinate objects that should be used instead.

xml_element(*doc*)
Returns a DOM element describing this coordinate factory.

property attributes

property climatological
Always returns False, as a factory itself can never have points/bounds and therefore can never be climatological by definition.

property coord_system
The coordinate-system (if any) of the coordinate made by the factory.

property dependencies
Returns a dictionary mapping from constructor argument names to the corresponding coordinates.

property long_name
The CF Metadata long name for the object.

property metadata

property standard_name
The CF Metadata standard name for the object.

property units
The S.I. unit of the object.

property var_name
The NetCDF variable name for the object.

Defines an ocean sigma coordinate factory.

class iris.aux_factory.**OceanSigmaFactory**(*sigma=None*, *eta=None*, *depth=None*)

Creates an ocean sigma coordinate factory with the formula:

z(n, k, j, i) = eta(n, j, i) + sigma(k) * (depth(j, i) + eta(n, j, i))

derived_dims(*coord_dims_func*)
Returns the cube dimensions for the derived coordinate.

Args:
- **coord_dims_func:** A callable which can return the list of dimensions relevant to a given coordinate. See *iris.cube.Cube.coord_dims()*.

Returns A sorted list of cube dimension numbers.

make_coord(*coord_dims_func*)
Returns a new *iris.coords.AuxCoord* as defined by this factory.

Args:
- **coord_dims_func:** A callable which can return the list of dimensions relevant to a given coordinate. See *iris.cube.Cube.coord_dims()*.

name(*default=None*, *token=False*)
Returns a string name representing the identity of the metadata.

First it tries standard name, then it tries the long name, then the NetCDF variable name, before falling-back to a default value, which itself defaults to the string 'unknown'.

Kwargs:
- **default:** The fall-back string representing the default name. Defaults to the string 'unknown'.
- **token:** If True, ensures that the name returned satisfies the criteria for the characters required by a valid NetCDF name. If it is not possible to return a valid name, then a ValueError exception is raised. Defaults to False.

Returns String.

rename(*name*)
Changes the human-readable name.

If 'name' is a valid standard name it will assign it to *standard_name*, otherwise it will assign it to *long_name*.

update(*old_coord*, *new_coord=None*)
Notifies the factory of the removal/replacement of a coordinate which might be a dependency.

Args:
- **old_coord:** The coordinate to be removed/replaced.
- **new_coord:** If None, any dependency using old_coord is removed, otherwise any dependency using old_coord is updated to use new_coord.

updated(*new_coord_mapping*)
Creates a new instance of this factory where the dependencies are replaced according to the given mapping.

Args:
- **new_coord_mapping:** A dictionary mapping from the object IDs potentially used by this factory, to the coordinate objects that should be used instead.

xml_element(*doc*)
Returns a DOM element describing this coordinate factory.

property attributes

property climatological
Always returns False, as a factory itself can never have points/bounds and therefore can never be climatological by definition.

property coord_system
The coordinate-system (if any) of the coordinate made by the factory.

property dependencies
Returns a dictionary mapping from constructor argument names to the corresponding coordinates.

property long_name
The CF Metadata long name for the object.

property metadata

property standard_name
The CF Metadata standard name for the object.

property units
The S.I. unit of the object.

property var_name
The NetCDF variable name for the object.

Defines an ocean sigma over z coordinate factory.

class iris.aux_factory.**OceanSigmaZFactory**(*sigma=None*, *eta=None*, *depth=None*, *depth_c=None*, *nsigma=None*, *zlev=None*)
Creates an ocean sigma over z coordinate factory with the formula:

if k < nsigma:

z(n, k, j, i) = eta(n, j, i) + sigma(k) * (min(depth_c, depth(j, i)) + eta(n, j, i))

if k >= nsigma: z(n, k, j, i) = zlev(k)

The *zlev* and 'nsigma' coordinates must be provided, and at least either *eta*, or 'sigma' and *depth* and *depth_c* coordinates.

derived_dims(*coord_dims_func*)
Returns the cube dimensions for the derived coordinate.

Args:
- **coord_dims_func:** A callable which can return the list of dimensions relevant to a given coordinate. See *iris.cube.Cube.coord_dims()*.

Returns A sorted list of cube dimension numbers.

make_coord(*coord_dims_func*)
Returns a new *iris.coords.AuxCoord* as defined by this factory.

Args:
- **coord_dims_func:** A callable which can return the list of dimensions relevant to a given coordinate. See *iris.cube.Cube.coord_dims()*.

name(*default=None*, *token=False*)
Returns a string name representing the identity of the metadata.

First it tries standard name, then it tries the long name, then the NetCDF variable name, before falling-back to a default value, which itself defaults to the string 'unknown'.

Kwargs:
- **default:** The fall-back string representing the default name. Defaults to the string 'unknown'.
- **token:** If True, ensures that the name returned satisfies the criteria for the characters required by a valid NetCDF name. If it is not possible to return a valid name, then a ValueError exception is raised. Defaults to False.

Returns String.

rename(*name*)
Changes the human-readable name.

If 'name' is a valid standard name it will assign it to *standard_name*, otherwise it will assign it to *long_name*.

update(*old_coord*, *new_coord=None*)
Notifies the factory of the removal/replacement of a coordinate which might be a dependency.

Args:
- **old_coord:** The coordinate to be removed/replaced.
- **new_coord:** If None, any dependency using old_coord is removed, otherwise any dependency using old_coord is updated to use new_coord.

updated(*new_coord_mapping*)
Creates a new instance of this factory where the dependencies are replaced according to the given mapping.

Args:
- **new_coord_mapping:** A dictionary mapping from the object IDs potentially used by this factory, to the coordinate objects that should be used instead.

xml_element(*doc*)
Returns a DOM element describing this coordinate factory.

property attributes

property climatological
Always returns False, as a factory itself can never have points/bounds and therefore can never be climatological by definition.

property coord_system
The coordinate-system (if any) of the coordinate made by the factory.

property dependencies
Returns a dictionary mapping from constructor argument names to the corresponding coordinates.

property long_name
The CF Metadata long name for the object.

property metadata

property standard_name
The CF Metadata standard name for the object.

property units
The S.I. unit of the object.

property var_name
The NetCDF variable name for the object.

27.3 iris.common

27.3.1 iris.common.lenient

Provides the infrastructure to support lenient client/service behaviour.

In this module:

- *LENIENT*
- *Lenient*

iris.common.lenient.**LENIENT**
(Public) Instance that manages all Iris run-time lenient features.

Thread-local data

class iris.common.lenient.**Lenient**(***kwargs*)
A container for managing the run-time lenient features and options.

Kwargs:
- **kwargs (dict)** Mapping of lenient key/value options to enable/disable. Note that, only the lenient "maths" options is available, which controls lenient/strict cube arithmetic.

For example:

```
Lenient(maths=False)
```

Note that, the values of these options are thread-specific.

context(***kwargs*)
Return a context manager which allows temporary modification of the lenient option state within the scope of the context manager.

On entry to the context manager, all provided keyword arguments are applied. On exit from the context manager, the previous lenient option state is restored.
For example::
with iris.common.Lenient.context(maths=False): pass

27.3.2 iris.common.metadata

Provides the infrastructure to support the common metadata API.

In this module:

- *SERVICES_COMBINE*
- *SERVICES_DIFFERENCE*
- *SERVICES_EQUAL*
- *SERVICES*
- *AncillaryVariableMetadata*
- *BaseMetadata*
- *CellMeasureMetadata*
- *CoordMetadata*
- *CubeMetadata*
- *DimCoordMetadata*
- *hexdigest*
- *metadata_manager_factory*

iris.common.metadata.**SERVICES_COMBINE**
Convenience collection of lenient metadata combine services.

---

iris.common.metadata.**SERVICES_DIFFERENCE**
Convenience collection of lenient metadata difference services.

---

iris.common.metadata.**SERVICES_EQUAL**
Convenience collection of lenient metadata equality services.

---

iris.common.metadata.**SERVICES**
Convenience collection of lenient metadata services.

Metadata container for a AncillaryVariableMetadata.

class iris.common.metadata.**AncillaryVariableMetadata**(*_cls*, *standard_name*, *long_name*, *var_name*, *units*, *attributes*)
Create new instance of AncillaryVariableMetadataNamedtuple(standard_name, long_name, var_name, units, attributes)

__eq__(*other*)
Determine whether the associated metadata members are equivalent.

Args:
- **other (metadata):** A metadata instance of the same type.

Returns Boolean.

combine(*other*, *lenient=None*)
Return a new metadata instance created by combining each of the associated metadata members.

Args:
- **other (metadata):** A metadata instance of the same type.

Kwargs:
- **lenient (boolean):** Enable/disable lenient combination. The default is to automatically detect whether this lenient operation is enabled.

Returns Metadata instance.

count(*value*, */*)
Return number of occurrences of value.

difference(*other*, *lenient=None*)
Return a new metadata instance created by performing a difference comparison between each of the associated metadata members.

A metadata member returned with a value of "None" indicates that there is no difference between the members being compared. Otherwise, a tuple of the different values is returned.

Args:
- **other (metadata):** A metadata instance of the same type.

Kwargs:
- **lenient (boolean):** Enable/disable lenient difference. The default is to automatically detect whether this lenient operation is enabled.

Returns Metadata instance of member differences or None.

equal(*other*, *lenient=None*)
Determine whether the associated metadata members are equivalent.

Args:
- **other (metadata):** A metadata instance of the same type.

Kwargs:
- **lenient (boolean):** Enable/disable lenient equivalence. The default is to automatically detect whether this lenient operation is enabled.

Returns Boolean.

classmethod from_metadata(*other*)
Convert the provided metadata instance from a different type to this metadata type, using only the relevant metadata members.

Non-common metadata members are set to `None`.

Args:
- **other (metadata):** A metadata instance of any type.

Returns New metadata instance.

index(*value*, *start=0*, *stop=9223372036854775807*, */*)
Return first index of value.

Raises ValueError if the value is not present.

name(*default=None*, *token=False*)
Returns a string name representing the identity of the metadata.

First it tries standard name, then it tries the long name, then the NetCDF variable name, before falling-back to a default value, which itself defaults to the string 'unknown'.

Kwargs:

- **default:** The fall-back string representing the default name. Defaults to the string 'unknown'.
- **token:** If True, ensures that the name returned satisfies the criteria for the characters required by a valid NetCDF name. If it is not possible to return a valid name, then a ValueError exception is raised. Defaults to False.

Returns String.

`classmethod token(name)`
Determine whether the provided name is a valid NetCDF name and thus safe to represent a single parsable token.

Args:

- **name:** The string name to verify

Returns The provided name if valid, otherwise None.

`DEFAULT_NAME = 'unknown'`

`attributes`
Alias for field number 4

`long_name`
Alias for field number 1

`standard_name`
Alias for field number 0

`units`
Alias for field number 3

`var_name`
Alias for field number 2

Container for common metadata.

`class iris.common.metadata.BaseMetadata(_cls, standard_name, long_name, var_name, units, attributes)`
Create new instance of BaseMetadataNamedtuple(standard_name, long_name, var_name, units, attributes)

`__eq__(other)`
Determine whether the associated metadata members are equivalent.

Args:

- **other (metadata):** A metadata instance of the same type.

Returns Boolean.

`combine(other, lenient=None)`
Return a new metadata instance created by combining each of the associated metadata members.

Args:

- **other (metadata):** A metadata instance of the same type.

Kwargs:

- **lenient (boolean):** Enable/disable lenient combination. The default is to automatically detect whether this lenient operation is enabled.

Returns Metadata instance.

`count`(*value*, */*)
Return number of occurrences of value.

`difference`(*other*, *lenient=None*)
Return a new metadata instance created by performing a difference comparison between each of the associated metadata members.

A metadata member returned with a value of "None" indicates that there is no difference between the members being compared. Otherwise, a tuple of the different values is returned.

Args:

- **other (metadata):** A metadata instance of the same type.

Kwargs:

- **lenient (boolean):** Enable/disable lenient difference. The default is to automatically detect whether this lenient operation is enabled.

Returns Metadata instance of member differences or None.

`equal`(*other*, *lenient=None*)
Determine whether the associated metadata members are equivalent.

Args:

- **other (metadata):** A metadata instance of the same type.

Kwargs:

- **lenient (boolean):** Enable/disable lenient equivalence. The default is to automatically detect whether this lenient operation is enabled.

Returns Boolean.

`classmethod from_metadata`(*other*)
Convert the provided metadata instance from a different type to this metadata type, using only the relevant metadata members.

Non-common metadata members are set to `None`.

Args:

- **other (metadata):** A metadata instance of any type.

Returns New metadata instance.

`index`(*value*, *start=0*, *stop=9223372036854775807*, */*)
Return first index of value.

Raises ValueError if the value is not present.

`name`(*default=None*, *token=False*)
Returns a string name representing the identity of the metadata.

First it tries standard name, then it tries the long name, then the NetCDF variable name, before falling-back to a default value, which itself defaults to the string 'unknown'.

Kwargs:

- **default:** The fall-back string representing the default name. Defaults to the string 'unknown'.

- **token:** If True, ensures that the name returned satisfies the criteria for the characters required by a valid NetCDF name. If it is not possible to return a valid name, then a ValueError exception is raised. Defaults to False.

Returns String.

classmethod token(*name*)
Determine whether the provided name is a valid NetCDF name and thus safe to represent a single parsable token.

Args:
- **name:** The string name to verify

Returns The provided name if valid, otherwise None.

DEFAULT_NAME = 'unknown'

attributes
Alias for field number 4

long_name
Alias for field number 1

standard_name
Alias for field number 0

units
Alias for field number 3

var_name
Alias for field number 2

Metadata container for a *CellMeasure*.

class iris.common.metadata.**CellMeasureMetadata**(*_cls*, *standard_name*, *long_name*, *var_name*, *units*, *attributes*, *measure*)
Create new instance of CellMeasureMetadataNamedtuple(standard_name, long_name, var_name, units, attributes, measure)

__eq__(*other*)
Determine whether the associated metadata members are equivalent.

Args:
- **other (metadata):** A metadata instance of the same type.

Returns Boolean.

combine(*other*, *lenient=None*)
Return a new metadata instance created by combining each of the associated metadata members.

Args:
- **other (metadata):** A metadata instance of the same type.

Kwargs:
- **lenient (boolean):** Enable/disable lenient combination. The default is to automatically detect whether this lenient operation is enabled.

Returns Metadata instance.

`count`(*value*, */*)
Return number of occurrences of value.

`difference`(*other*, *lenient=None*)
Return a new metadata instance created by performing a difference comparison between each of the associated metadata members.

A metadata member returned with a value of "None" indicates that there is no difference between the members being compared. Otherwise, a tuple of the different values is returned.

Args:
- **other (metadata):** A metadata instance of the same type.

Kwargs:
- **lenient (boolean):** Enable/disable lenient difference. The default is to automatically detect whether this lenient operation is enabled.

Returns Metadata instance of member differences or None.

`equal`(*other*, *lenient=None*)
Determine whether the associated metadata members are equivalent.

Args:
- **other (metadata):** A metadata instance of the same type.

Kwargs:
- **lenient (boolean):** Enable/disable lenient equivalence. The default is to automatically detect whether this lenient operation is enabled.

Returns Boolean.

`classmethod from_metadata`(*other*)
Convert the provided metadata instance from a different type to this metadata type, using only the relevant metadata members.

Non-common metadata members are set to `None`.

Args:
- **other (metadata):** A metadata instance of any type.

Returns New metadata instance.

`index`(*value*, *start=0*, *stop=9223372036854775807*, */*)
Return first index of value.

Raises ValueError if the value is not present.

`name`(*default=None*, *token=False*)
Returns a string name representing the identity of the metadata.

First it tries standard name, then it tries the long name, then the NetCDF variable name, before falling-back to a default value, which itself defaults to the string 'unknown'.

Kwargs:
- **default:** The fall-back string representing the default name. Defaults to the string 'unknown'.
- **token:** If True, ensures that the name returned satisfies the criteria for the characters required by a valid NetCDF name. If it is not possible to return a valid name, then a ValueError exception is raised. Defaults to False.

Returns String.

classmethod token(*name*)
Determine whether the provided name is a valid NetCDF name and thus safe to represent a single parsable token.

Args:
- **name:** The string name to verify

Returns The provided name if valid, otherwise None.

DEFAULT_NAME = 'unknown'

attributes
Alias for field number 4

long_name
Alias for field number 1

measure
Alias for field number 5

standard_name
Alias for field number 0

units
Alias for field number 3

var_name
Alias for field number 2

Metadata container for a *Coord*.

class iris.common.metadata.**CoordMetadata**(*_cls*, *standard_name*, *long_name*, *var_name*, *units*, *attributes*, *coord_system*, *climatological*)
Create new instance of CoordMetadataNamedtuple(standard_name, long_name, var_name, units, attributes, coord_system, climatological)

__eq__(*other*)
Determine whether the associated metadata members are equivalent.

Args:
- **other (metadata):** A metadata instance of the same type.

Returns Boolean.

combine(*other*, *lenient=None*)
Return a new metadata instance created by combining each of the associated metadata members.

Args:
- **other (metadata):** A metadata instance of the same type.

Kwargs:
- **lenient (boolean):** Enable/disable lenient combination. The default is to automatically detect whether this lenient operation is enabled.

Returns Metadata instance.

count(*value*, /)
Return number of occurrences of value.

difference(*other*, *lenient=None*)
Return a new metadata instance created by performing a difference comparison between each of the associated metadata members.

A metadata member returned with a value of "None" indicates that there is no difference between the members being compared. Otherwise, a tuple of the different values is returned.

Args:
- **other (metadata):** A metadata instance of the same type.

Kwargs:
- **lenient (boolean):** Enable/disable lenient difference. The default is to automatically detect whether this lenient operation is enabled.

Returns Metadata instance of member differences or None.

equal(*other*, *lenient=None*)
Determine whether the associated metadata members are equivalent.

Args:
- **other (metadata):** A metadata instance of the same type.

Kwargs:
- **lenient (boolean):** Enable/disable lenient equivalence. The default is to automatically detect whether this lenient operation is enabled.

Returns Boolean.

classmethod from_metadata(*other*)
Convert the provided metadata instance from a different type to this metadata type, using only the relevant metadata members.

Non-common metadata members are set to `None`.

Args:
- **other (metadata):** A metadata instance of any type.

Returns New metadata instance.

index(*value*, *start=0*, *stop=9223372036854775807*, /)
Return first index of value.

Raises ValueError if the value is not present.

name(*default=None*, *token=False*)
Returns a string name representing the identity of the metadata.

First it tries standard name, then it tries the long name, then the NetCDF variable name, before falling-back to a default value, which itself defaults to the string 'unknown'.

Kwargs:
- **default:** The fall-back string representing the default name. Defaults to the string 'unknown'.
- **token:** If True, ensures that the name returned satisfies the criteria for the characters required by a valid NetCDF name. If it is not possible to return a valid name, then a ValueError exception is raised. Defaults to False.

Returns String.

classmethod token(*name*)
Determine whether the provided name is a valid NetCDF name and thus safe to represent a single parsable token.

Args:
- **name:** The string name to verify

Returns The provided name if valid, otherwise None.

DEFAULT_NAME = 'unknown'

attributes
Alias for field number 4

climatological
Alias for field number 6

coord_system
Alias for field number 5

long_name
Alias for field number 1

standard_name
Alias for field number 0

units
Alias for field number 3

var_name
Alias for field number 2

Metadata container for a *Cube*.

class iris.common.metadata.**CubeMetadata**(*_cls*, *standard_name*, *long_name*, *var_name*, *units*, *attributes*, *cell_methods*)
Create new instance of CubeMetadataNamedtuple(standard_name, long_name, var_name, units, attributes, cell_methods)

__eq__(*other*)
Determine whether the associated metadata members are equivalent.

Args:
- **other (metadata):** A metadata instance of the same type.

Returns Boolean.

combine(*other*, *lenient=None*)
Return a new metadata instance created by combining each of the associated metadata members.

Args:
- **other (metadata):** A metadata instance of the same type.

Kwargs:
- **lenient (boolean):** Enable/disable lenient combination. The default is to automatically detect whether this lenient operation is enabled.

Returns Metadata instance.

count(*value*, */*)
Return number of occurrences of value.

`difference`(*other*, *lenient=None*)
Return a new metadata instance created by performing a difference comparison between each of the associated metadata members.

A metadata member returned with a value of "None" indicates that there is no difference between the members being compared. Otherwise, a tuple of the different values is returned.

Args:
- **other (metadata):** A metadata instance of the same type.

Kwargs:
- **lenient (boolean):** Enable/disable lenient difference. The default is to automatically detect whether this lenient operation is enabled.

Returns Metadata instance of member differences or None.

`equal`(*other*, *lenient=None*)
Determine whether the associated metadata members are equivalent.

Args:
- **other (metadata):** A metadata instance of the same type.

Kwargs:
- **lenient (boolean):** Enable/disable lenient equivalence. The default is to automatically detect whether this lenient operation is enabled.

Returns Boolean.

`classmethod from_metadata`(*other*)
Convert the provided metadata instance from a different type to this metadata type, using only the relevant metadata members.

Non-common metadata members are set to `None`.

Args:
- **other (metadata):** A metadata instance of any type.

Returns New metadata instance.

`index`(*value*, *start=0*, *stop=9223372036854775807*, */*)
Return first index of value.

Raises ValueError if the value is not present.

`name`(*default=None*, *token=False*)
Returns a string name representing the identity of the metadata.

First it tries standard name, then it tries the long name, then the NetCDF variable name, before falling-back to a default value, which itself defaults to the string 'unknown'.

Kwargs:
- **default:** The fall-back string representing the default name. Defaults to the string 'unknown'.
- **token:** If True, ensures that the name returned satisfies the criteria for the characters required by a valid NetCDF name. If it is not possible to return a valid name, then a ValueError exception is raised. Defaults to False.

Returns String.

classmethod token(*name*)
Determine whether the provided name is a valid NetCDF name and thus safe to represent a single parsable token.

Args:
- **name:** The string name to verify

Returns The provided name if valid, otherwise None.

DEFAULT_NAME = 'unknown'

attributes
Alias for field number 4

cell_methods
Alias for field number 5

long_name
Alias for field number 1

standard_name
Alias for field number 0

units
Alias for field number 3

var_name
Alias for field number 2

Metadata container for a *DimCoord*

class iris.common.metadata.**DimCoordMetadata**(*_cls*, *standard_name*, *long_name*, *var_name*, *units*, *attributes*, *coord_system*, *climatological*, *circular*)
Create new instance of DimCoordMetadataNamedtuple(standard_name, long_name, var_name, units, attributes, coord_system, climatological, circular)

__eq__(*other*)
Determine whether the associated metadata members are equivalent.

Args:
- **other (metadata):** A metadata instance of the same type.

Returns Boolean.

combine(*other*, *lenient=None*)
Return a new metadata instance created by combining each of the associated metadata members.

Args:
- **other (metadata):** A metadata instance of the same type.

Kwargs:
- **lenient (boolean):** Enable/disable lenient combination. The default is to automatically detect whether this lenient operation is enabled.

Returns Metadata instance.

count(*value*, /)
Return number of occurrences of value.

difference(*other*, *lenient=None*)
Return a new metadata instance created by performing a difference comparison between each of the associated metadata members.

A metadata member returned with a value of "None" indicates that there is no difference between the members being compared. Otherwise, a tuple of the different values is returned.

Args:
- **other (metadata):** A metadata instance of the same type.

Kwargs:
- **lenient (boolean):** Enable/disable lenient difference. The default is to automatically detect whether this lenient operation is enabled.

Returns Metadata instance of member differences or None.

equal(*other*, *lenient=None*)
Determine whether the associated metadata members are equivalent.

Args:
- **other (metadata):** A metadata instance of the same type.

Kwargs:
- **lenient (boolean):** Enable/disable lenient equivalence. The default is to automatically detect whether this lenient operation is enabled.

Returns Boolean.

classmethod from_metadata(*other*)
Convert the provided metadata instance from a different type to this metadata type, using only the relevant metadata members.

Non-common metadata members are set to `None`.

Args:
- **other (metadata):** A metadata instance of any type.

Returns New metadata instance.

index(*value*, *start=0*, *stop=9223372036854775807*, /)
Return first index of value.

Raises ValueError if the value is not present.

name(*default=None*, *token=False*)
Returns a string name representing the identity of the metadata.

First it tries standard name, then it tries the long name, then the NetCDF variable name, before falling-back to a default value, which itself defaults to the string 'unknown'.

Kwargs:
- **default:** The fall-back string representing the default name. Defaults to the string 'unknown'.
- **token:** If True, ensures that the name returned satisfies the criteria for the characters required by a valid NetCDF name. If it is not possible to return a valid name, then a ValueError exception is raised. Defaults to False.

Returns String.

classmethod token(*name*)
Determine whether the provided name is a valid NetCDF name and thus safe to represent a single parsable token.

Args:
- **name:** The string name to verify

Returns The provided name if valid, otherwise None.

DEFAULT_NAME = 'unknown'

attributes
Alias for field number 4

circular
Alias for field number 7

climatological
Alias for field number 6

coord_system
Alias for field number 5

long_name
Alias for field number 1

standard_name
Alias for field number 0

units
Alias for field number 3

var_name
Alias for field number 2

---

iris.common.metadata.**hexdigest**(*item*)
Calculate a hexidecimal string hash representation of the provided item.

Calculates a 64-bit non-cryptographic hash of the provided item, using the extremely fast `xxhash` hashing algorithm, and returns the hexdigest string representation of the hash.

This provides a means to compare large and/or complex objects through simple string hexdigest comparison.

Args:

- **item (object):** The item that requires to have its hexdigest calculated.

Returns The string hexidecimal representation of the item's 64-bit hash.

---

iris.common.metadata.**metadata_manager_factory**(*cls*, ***kwargs*)
A class instance factory function responsible for manufacturing metadata instances dynamically at runtime.

The factory instances returned by the factory are capable of managing their metadata state, which can be proxied by the owning container.

Args:

- **cls:** A subclass of *BaseMetadata*, defining the metadata to be managed.

Kwargs:

- **kwargs:** Initial values for the manufactured metadata instance. Unspecified fields will default to a value of 'None'.

Returns A manager instance for the provided metadata `cls`.

27.3.3 iris.common.mixin

Provides common metadata mixin behaviour.

In this module:

- *CFVariableMixin*

None

class iris.common.mixin.**CFVariableMixin**

name(*default=None*, *token=False*)
Returns a string name representing the identity of the metadata.

First it tries standard name, then it tries the long name, then the NetCDF variable name, before falling-back to a default value, which itself defaults to the string 'unknown'.

Kwargs:
- **default:** The fall-back string representing the default name. Defaults to the string 'unknown'.
- **token:** If True, ensures that the name returned satisfies the criteria for the characters required by a valid NetCDF name. If it is not possible to return a valid name, then a ValueError exception is raised. Defaults to False.

Returns String.

rename(*name*)
Changes the human-readable name.

If 'name' is a valid standard name it will assign it to *standard_name*, otherwise it will assign it to *long_name*.

property attributes

property long_name
The CF Metadata long name for the object.

property metadata

property standard_name
The CF Metadata standard name for the object.

property units
The S.I. unit of the object.

property var_name
The NetCDF variable name for the object.

27.3.4 iris.common.resolve

Provides the infrastructure to support the analysis, identification and combination of metadata common between two *Cube* operands into a single resultant *Cube*, which will be auto-transposed, and with the appropriate broadcast shape.

In this module:

- *Resolve*

At present, *Resolve* is used by Iris solely during cube maths to combine a left-hand *Cube* operand and a right-hand *Cube* operand into a resultant *Cube* with common metadata, suitably auto-transposed dimensions, and an appropriate broadcast shape.

However, the capability and benefit provided by *Resolve* may be exercised as a general means to easily and consistently combine the metadata of two *Cube* operands together into a single resultant *Cube*. This is highlighted through the following use case patterns.

Firstly, creating a `resolver` instance with *specific* *Cube* operands, and then supplying `data` with suitable dimensionality and shape to create the resultant resolved *Cube*, e.g.,

```
>>> print(cube1)
air_temperature / (K)               (time: 240; latitude: 37;␣
↪longitude: 49)
    Dimension coordinates:
        time                            x              -            ␣
↪ -
        latitude                        -              x            ␣
↪ -
        longitude                       -              -            ␣
↪ x
    Auxiliary coordinates:
        forecast_period                 x              -            ␣
↪ -
    Scalar coordinates:
        forecast_reference_time     1859-09-01 06:00:00
        height                      1.5 m
    Cell methods:
        mean                        time (6 hour)
    Attributes:
        Conventions                 CF-1.5
        Model scenario              A1B
        STASH                       m01s03i236
        source                      Data from Met Office Unified␣
↪Model 6.05

>>> print(cube2)
air_temperature / (K)               (longitude: 49; latitude: 37)
    Dimension coordinates:
        longitude                           x              -
        latitude                            -              x
    Scalar coordinates:
        forecast_period             10794 hours
        forecast_reference_time     1859-09-01 06:00:00
        height                      1.5 m
        time                        1860-06-01 00:00:00, bound=(1859-
↪12-01 00:00:00, 1860-12-01 00:00:00)
```

(continues on next page)

(continued from previous page)

```
    Cell methods:
        mean                        time (6 hour)
    Attributes:
        Conventions                 CF-1.5
        Model scenario              E1
        STASH                       m01s03i236
        source                      Data from Met Office Unified␣
↪Model 6.05

>>> print(data.shape)
(240, 37, 49)
>>> resolver = Resolve(cube1, cube2)
>>> result = resolver.cube(data)
>>> print(result)
air_temperature / (K)               (time: 240; latitude: 37;␣
↪longitude: 49)
    Dimension coordinates:
        time                            x              -           ␣
↪ -
        latitude                        -              x           ␣
↪ -
        longitude                       -              -           ␣
↪ x
    Auxiliary coordinates:
        forecast_period                 x              -           ␣
↪ -
    Scalar coordinates:
        forecast_reference_time     1859-09-01 06:00:00
        height                      1.5 m
    Cell methods:
        mean                        time (6 hour)
    Attributes:
        Conventions                 CF-1.5
        STASH                       m01s03i236
        source                      Data from Met Office Unified␣
↪Model 6.05
```

Secondly, creating an *empty* `resolver` instance, that may be called *multiple* times with *different* *Cube* operands and *different* `data`, e.g.,

```
>>> resolver = Resolve()
>>> result1 = resolver(cube1, cube2).cube(data1)
>>> result2 = resolver(cube3, cube4).cube(data2)
```

Lastly, creating a `resolver` instance with *specific* *Cube* operands, and then supply *different* `data` *multiple* times, e.g.,

```
>>> payload = (data1, data2, data3)
>>> resolver = Resolve(cube1, cube2)
>>> results = [resolver.cube(data) for data in payload]
```

class iris.common.resolve.**Resolve**(*lhs=None*, *rhs=None*)

Resolve the provided `lhs` *Cube* operand and `rhs` *Cube* operand to determine

the metadata that is common between them, and the auto-transposed, broadcast shape of the resultant *Cube*.

This includes the identification of common *CubeMetadata*, *DimCoord*, *AuxCoord*, and *AuxCoordFactory* metadata.

Note: Resolving common *AncillaryVariable* and *CellMeasure* metadata is not supported at this time. (Issue #3839)

Note: A *Resolve* instance is **callable**, allowing two new `lhs` and `rhs` *Cube* operands to be resolved. Note that, *Resolve* only supports resolving **two** operands at a time, and no more.

Warning: *Resolve* attempts to preserve commutativity, but this may not be possible when auto-transposition or extended broadcasting is involved during the operation.

For example:

```
>>> cube1
<iris 'Cube' of air_temperature / (K) (time: 240;␣
↪latitude: 37; longitude: 49)>
>>> cube2
<iris 'Cube' of air_temperature / (K) (longitude: 49;␣
↪latitude: 37)>
>>> result1 = Resolve(cube1, cube2).cube(data)
>>> result2 = Resolve(cube2, cube1).cube(data)
>>> result1 == result2
True
```

Kwargs:

- **lhs:** The left-hand-side *Cube* operand.
- **rhs:** The right-hand-side *Cube* operand.

__call__(*lhs*, *rhs*)

Resolve the `lhs` *Cube* operand and `rhs` *Cube* operand metadata.

Involves determining all the common coordinate metadata shared between the operands, and the metadata that is local to each operand. Given the common metadata, the broadcast shape of the resultant resolved *Cube*, which may be auto-transposed, can be determined.

Args:

- **lhs:** The left-hand-side *Cube* operand.
- **rhs:** The right-hand-side *Cube* operand.

cube(*data*, *in_place=False*)

Create the resultant *Cube* from the resolved `lhs` and `rhs` *Cube* operands, using the provided `data`.

Args:

- **data:** The data payload for the resultant *Cube*, which **must match** the expected resolved *shape*.

Kwargs:

- **in_place:** If `True`, the `data` is inserted into the `tgt` *Cube*. The existing metadata of the `tgt` *Cube* is replaced with the resolved metadata from the `lhs` and `rhs` *Cube* operands. Otherwise, a **new** *Cube* instance is returned. Default is `False`.

Returns *Cube*

Note: *Resolve* will determine whether the `lhs` *Cube* operand is mapped to the `rhs` *Cube* operand, or vice versa. In general, the **lower rank** operand (`src`) is mapped to the **higher rank** operand (`tgt`). Therefore, the `src` *Cube* may be either the `lhs` or the `rhs` *Cube* operand, given the direction of the mapping. See *map_rhs_to_lhs*.

Warning: It may not be possible to perform an `in_place` operation, due to any transposition or extended broadcasting that requires to be performed i.e., the `tgt` *Cube* **must match** the expected resolved *shape*.

For example:

```
>>> resolver = Resolve(cube1, cube2)
>>> resolver.map_rhs_to_lhs
True
>>> cube1.data.sum()
124652160.0
>>> zeros.shape
(240, 37, 49)
>>> zeros.sum()
0.0
>>> result = resolver.cube(zeros, in_place=True)
>>> result is cube1
True
>>> cube1.data.sum()
0.0
```

category_common
Categorised dim, aux and scalar coordinate items **common** to both the `lhs` *Cube* and the `rhs` *Cube*.

lhs_cube
The `lhs` operand to be resolved into the resultant *Cube*.

lhs_cube_aux_coverage
Analysis of aux and scalar coordinates spanning the `lhs` *Cube*.

lhs_cube_category
Categorised dim, aux and scalar coordinate items for `lhs` *Cube*.

lhs_cube_category_local
Categorised dim, aux and scalar coordinate items **local** to the `lhs` *Cube* only.

lhs_cube_dim_coverage
Analysis of dim coordinates spanning the `lhs` *Cube*.

lhs_cube_resolved
The transposed/reshaped (if required) `lhs` *Cube*, which can be broadcast with the `rhs` *Cube*.

map_rhs_to_lhs
Map **common** metadata from the `rhs` *Cube* to the `lhs` *Cube* if `lhs.rank >= rhs.rank`, otherwise map **common** metadata from the `lhs` *Cube* to the `rhs` *Cube*.

property mapped
Boolean state representing whether **all** `src` *Cube* dimensions have been associated with relevant `tgt` *Cube* dimensions.

Note: *Resolve* will determine whether the `lhs` *Cube* operand is mapped to the `rhs` *Cube* operand, or vice versa. In general, the **lower rank** operand (`src`) is mapped to the **higher rank** operand (`tgt`). Therefore, the `src` *Cube* may be either the `lhs` or the `rhs` *Cube* operand, given the direction of the mapping. See *map_rhs_to_lhs*.

If no *Cube* operands have been provided, then `mapped` is `None`.

For example:

```
>>> print(cube1)
air_temperature / (K)               (time: 240;␣
↪latitude: 37; longitude: 49)
    Dimension coordinates:
        time                             x            ␣
↪ -              -
        latitude                         -            ␣
↪ x              -
        longitude                        -            ␣
↪ -              x
    Auxiliary coordinates:
        forecast_period                  x            ␣
↪ -              -
    Scalar coordinates:
        forecast_reference_time     1859-09-01 06:00:00
        height                      1.5 m
    Cell methods:
        mean                        time (6 hour)
    Attributes:
        Conventions                 CF-1.5
        Model scenario              A1B
        STASH                       m01s03i236
        source                      Data from Met␣
↪Office Unified Model 6.05
>>> print(cube2)
air_temperature / (K)               (longitude: 49;␣
↪latitude: 37)
    Dimension coordinates:
```

(continues on next page)

(continued from previous page)

```
        longitude                               x       ␣
↪     -
        latitude                                -       ␣
↪     x
    Scalar coordinates:
        forecast_period             10794 hours
        forecast_reference_time     1859-09-01 06:00:00
        height                      1.5 m
        time                        1860-06-01␣
↪00:00:00, bound=(1859-12-01 00:00:00, 1860-12-01␣
↪00:00:00)
    Cell methods:
        mean                        time (6 hour)
    Attributes:
        Conventions                 CF-1.5
        Model scenario              E1
        STASH                       m01s03i236
        source                      Data from Met␣
↪Office Unified Model 6.05
>>> Resolve().mapped is None
True
>>> resolver = Resolve(cube1, cube2)
>>> resolver.mapped
True
>>> resolver.map_rhs_to_lhs
True
>>> resolver = Resolve(cube2, cube1)
>>> resolver.mapped
True
>>> resolver.map_rhs_to_lhs
False
```

mapping
Mapping of the dimensions between **common** metadata for the *Cube* operands, where the direction of the mapping is governed by *map_rhs_to_lhs*.

prepared_category
Cache containing a list of dim, aux and scalar coordinates prepared and ready for creating and attaching to the resultant resolved *Cube*.

prepared_factories
Cache containing a list of aux factories prepared and ready for creating and attaching to the resultant resolved *Cube*.

rhs_cube
The `rhs` operand to be resolved into the resultant *Cube*.

rhs_cube_aux_coverage
Analysis of aux and scalar coordinates spanning the `rhs` *Cube*.

rhs_cube_category
Categorised dim, aux and scalar coordinate items for `rhs` *Cube*.

rhs_cube_category_local
Categorised dim, aux and scalar coordinate items **local** to the `rhs` *Cube*

only.

rhs_cube_dim_coverage
Analysis of dim coordinates spanning the `rhs` *Cube*.

rhs_cube_resolved
The transposed/reshaped (if required) `rhs` *Cube*, which can be broadcast with the `lhs` *Cube*.

property shape
Proposed shape of the final resolved cube given the `lhs` *Cube* operand and the `rhs` *Cube* operand.

If no *Cube* operands have been provided, then `shape` is `None`.

For example:

```
>>> print(cube1)
air_temperature / (K)               (time: 240;␣
↪latitude: 37; longitude: 49)
    Dimension coordinates:
        time                             x            ␣
↪ -              -
        latitude                         -            ␣
↪ x              -
        longitude                        -            ␣
↪ -              x
    Auxiliary coordinates:
        forecast_period                  x            ␣
↪ -              -
    Scalar coordinates:
        forecast_reference_time     1859-09-01 06:00:00
        height                      1.5 m
    Cell methods:
        mean                        time (6 hour)
    Attributes:
        Conventions                 CF-1.5
        Model scenario              A1B
        STASH                       m01s03i236
        source                      Data from Met␣
↪Office Unified Model 6.05
>>> print(cube2)
air_temperature / (K)               (longitude: 49;␣
↪latitude: 37)
    Dimension coordinates:
        longitude                            x       ␣
↪     -
        latitude                             -       ␣
↪     x
    Scalar coordinates:
        forecast_period             10794 hours
        forecast_reference_time     1859-09-01 06:00:00
        height                      1.5 m
        time                        1860-06-01␣
↪00:00:00, bound=(1859-12-01 00:00:00, 1860-12-01␣
↪00:00:00)
```

(continues on next page)

(continued from previous page)

```
    Cell methods:
        mean                            time (6 hour)
    Attributes:
        Conventions                     CF-1.5
        Model scenario                  E1
        STASH                           m01s03i236
        source                          Data from Met␣
↪Office Unified Model 6.05
>>> Resolve().shape is None
True
>>> Resolve(cube1, cube2).shape
(240, 37, 49)
>>> Resolve(cube2, cube1).shape
(240, 37, 49)
```

A package for provisioning common Iris infrastructure.

In this module:

27.4 iris.config

Provides access to Iris-specific configuration values.

The default configuration values can be overridden by creating the file `iris/etc/site.cfg`. If it exists, this file must conform to the format defined by `ConfigParser`.

`iris.config.`**TEST_DATA_DIR**
Local directory where test data exists. Defaults to "test_data" sub-directory of the Iris package install directory. The test data directory supports the subset of Iris unit tests that require data. Directory contents accessed via `iris.tests.get_data_path()`.

`iris.config.`**PALETTE_PATH**
The full path to the Iris palette configuration directory

`iris.config.`**IMPORT_LOGGER**
The [optional] name of the logger to notify when first imported.

In this module:

- *netcdf*
- *get_dir_option*
- *get_logger*
- *get_option*
- *NetCDF*

`iris.config.`**netcdf**
Control Iris NetCDF options.

---

iris.config.**get_dir_option**(*section*, *option*, *default=None*)
Returns the directory path from the given option and section, or returns the given default value if the section/option is not present or does not represent a valid directory.

---

iris.config.**get_logger**(*name*, *datefmt=None*, *fmt=None*, *level=None*, *propagate=None*)
Create a `logging.Logger` with a `logging.StreamHandler` and custom `logging.Formatter`.

Args:

- **name:** The name of the logger. Typically this is the module filename that owns the logger.

Kwargs:

- **datefmt:** The date format string of the `logging.Formatter`. Defaults to `%d-%m-%Y %H:%M:%S`.
- **fmt:** The additional format string of the `logging.Formatter`. This is appended to the default format string `%(asctime)s %(name)s %(levelname)s - %(message)s`.
- **level:** The threshold level of the logger. Defaults to `INFO`.
- **propagate:** Sets the `propagate` attribute of the `logging.Logger`, which determines whether events logged to this logger will be passed to the handlers of higher level loggers. Defaults to `False`.

---

iris.config.**get_option**(*section*, *option*, *default=None*)
Returns the option value for the given section, or the default value if the section/option is not present.

Control Iris NetCDF options.

class iris.config.**NetCDF**(*conventions_override=None*)
Set up NetCDF processing options for Iris.

Currently accepted kwargs:

- **conventions_override (bool):** Define whether the CF Conventions version (e.g. *CF-1.6*) set when saving a cube to a NetCDF file should be defined by Iris (the default) or the cube being saved.

  If *False* (the default), specifies that Iris should set the CF Conventions version when saving cubes as NetCDF files. If *True*, specifies that the cubes being saved to NetCDF should set the CF Conventions version for the saved NetCDF files.

Example usages:

- Specify, for the lifetime of the session, that we want all cubes written to NetCDF to define their own CF Conventions versions:

```
iris.config.netcdf.conventions_override = True
iris.save('my_cube', 'my_dataset.nc')
iris.save('my_second_cube', 'my_second_dataset.nc')
```

- Specify, with a context manager, that we want a cube written to NetCDF to define its own CF Conventions version:

```
with iris.config.netcdf.context(conventions_override=True):
    iris.save('my_cube', 'my_dataset.nc')
```

context(***kwargs*)
Allow temporary modification of the options via a context manager. Accepted kwargs are the same as can be supplied to the Option.

27.5 iris.coord_categorisation

Cube functions for coordinate categorisation.

All the functions provided here add a new coordinate to a cube.

- The function *add_categorised_coord()* performs a generic coordinate categorisation.
- The other functions all implement specific common cases (e.g. *add_day_of_month()*). Currently, these are all calendar functions, so they only apply to "Time coordinates".

In this module:

- *add_categorised_coord*
- *add_day_of_month*
- *add_day_of_year*
- *add_hour*
- *add_month*
- *add_month_fullname*
- *add_month_number*
- *add_season*
- *add_season_membership*
- *add_season_number*
- *add_season_year*
- *add_weekday*
- *add_weekday_fullname*
- *add_weekday_number*
- *add_year*

iris.coord_categorisation.**add_categorised_coord**(*cube*, *name*, *from_coord*, *category_function*, *units='1'*)
Add a new coordinate to a cube, by categorising an existing one.

Make a new *iris.coords.AuxCoord* from mapped values, and add it to the cube.

Args:

- **cube (*iris.cube.Cube*):** the cube containing 'from_coord'. The new coord will be added into it.
- **name (string):** name of the created coordinate
- **from_coord (*iris.coords.Coord* or string):** coordinate in 'cube', or the name of one
- **category_function (callable):** function(coordinate, value), returning a category value for a coordinate point-value

Kwargs:

- **units:** units of the category value, typically ‘no_unit’ or ‘1’.

---

iris.coord_categorisation.**add_day_of_month**(*cube*, *coord*, *name='day_of_month'*)
Add a categorical day-of-month coordinate, values 1..31.

---

iris.coord_categorisation.**add_day_of_year**(*cube*, *coord*, *name='day_of_year'*)
Add a categorical day-of-year coordinate, values 1..365 (1..366 in leap years).

---

iris.coord_categorisation.**add_hour**(*cube*, *coord*, *name='hour'*)
Add a categorical hour coordinate, values 0..23.

---

iris.coord_categorisation.**add_month**(*cube*, *coord*, *name='month'*)
Add a categorical month coordinate, values ‘Jan’..’Dec’.

---

iris.coord_categorisation.**add_month_fullname**(*cube*, *coord*, *name='month_fullname'*)
Add a categorical month coordinate, values ‘January’..’December’.

---

iris.coord_categorisation.**add_month_number**(*cube*, *coord*, *name='month_number'*)
Add a categorical month coordinate, values 1..12.

---

iris.coord_categorisation.**add_season**(*cube*, *coord*, *name='season'*, *seasons=('djf', 'mam', 'jja', 'son')*)
Add a categorical season-of-year coordinate, with user specified seasons.

Args:

- **cube (*iris.cube.Cube*):** The cube containing ‘coord’. The new coord will be added into it.
- **coord (*iris.coords.Coord* or string):** Coordinate in ‘cube’, or its name, representing time.

Kwargs:

- **name (string):** Name of the created coordinate. Defaults to “season”.
- **seasons (list of strings):** List of seasons defined by month abbreviations. Each month must appear once and only once. Defaults to standard meteorological seasons (‘djf’, ‘mam’, ‘jja’, ‘son’).

---

iris.coord_categorisation.**add_season_membership**(*cube*, *coord*, *season*, *name='season_membership'*)
Add a categorical season membership coordinate for a user specified season.

The coordinate has the value True for every time that is within the given season, and the value False otherwise.

Args:

- **cube (*iris.cube.Cube*):** The cube containing 'coord'. The new coord will be added into it.
- **coord (*iris.coords.Coord* or string):** Coordinate in 'cube', or its name, representing time.
- **season (string):** Season defined by month abbreviations.

Kwargs:

- **name (string):** Name of the created coordinate. Defaults to "season_membership".

---

iris.coord_categorisation.**add_season_number**(*cube*, *coord*, *name='season_number'*, *seasons=('djf', 'mam', 'jja', 'son')*)

Add a categorical season-of-year coordinate, values 0..N-1 where N is the number of user specified seasons.

Args:

- **cube (*iris.cube.Cube*):** The cube containing 'coord'. The new coord will be added into it.
- **coord (*iris.coords.Coord* or string):** Coordinate in 'cube', or its name, representing time.

Kwargs:

- **name (string):** Name of the created coordinate. Defaults to "season_number".
- **seasons (list of strings):** List of seasons defined by month abbreviations. Each month must appear once and only once. Defaults to standard meteorological seasons ('djf', 'mam', 'jja', 'son').

---

iris.coord_categorisation.**add_season_year**(*cube*, *coord*, *name='season_year'*, *seasons=('djf', 'mam', 'jja', 'son')*)

Add a categorical year-of-season coordinate, with user specified seasons.

Args:

- **cube (*iris.cube.Cube*):** The cube containing 'coord'. The new coord will be added into it.
- **coord (*iris.coords.Coord* or string):** Coordinate in 'cube', or its name, representing time.

Kwargs:

- **name (string):** Name of the created coordinate. Defaults to "season_year".
- **seasons (list of strings):** List of seasons defined by month abbreviations. Each month must appear once and only once. Defaults to standard meteorological seasons ('djf', 'mam', 'jja', 'son').

---

iris.coord_categorisation.**add_weekday**(*cube*, *coord*, *name='weekday'*)

Add a categorical weekday coordinate, values 'Mon'..'Sun'.

---

iris.coord_categorisation.**add_weekday_fullname**(*cube*, *coord*, *name='weekday_fullname'*)

Add a categorical weekday coordinate, values 'Monday'..'Sunday'.

---

iris.coord_categorisation.**add_weekday_number**(*cube*, *coord*, *name='weekday_number'*)
Add a categorical weekday coordinate, values 0..6 [0=Monday].

---

iris.coord_categorisation.**add_year**(*cube*, *coord*, *name='year'*)
Add a categorical calendar-year coordinate.

27.6 iris.coord_systems

Definitions of coordinate systems.

In this module:

- *AlbersEqualArea*
- *CoordSystem*
- *GeogCS*
- *Geostationary*
- *LambertAzimuthalEqualArea*
- *LambertConformal*
- *Mercator*
- *OSGB*
- *Orthographic*
- *RotatedGeogCS*
- *Stereographic*
- *TransverseMercator*
- *VerticalPerspective*

A coordinate system in the Albers Conical Equal Area projection.

class iris.coord_systems.**AlbersEqualArea**(*latitude_of_projection_origin=None*, *longitude_of_central_meridian=None*, *false_easting=None*, *false_northing=None*, *standard_parallels=None*, *ellipsoid=None*)

Constructs a Albers Conical Equal Area coord system.

Kwargs:

- **latitude_of_projection_origin:** True latitude of planar origin in degrees. Defaults to 0.0 .
- **longitude_of_central_meridian:** True longitude of planar central meridian in degrees. Defaults to 0.0 .
- **false_easting:** X offset from planar origin in metres. Defaults to 0.0 .
- **false_northing:** Y offset from planar origin in metres. Defaults to 0.0 .

- **standard_parallels (number or iterable of 1 or 2 numbers):** The one or two latitudes of correct scale. Defaults to (20.0, 50.0).
- **ellipsoid (*GeogCS*):** If given, defines the ellipsoid.

as_cartopy_crs()
Return a cartopy CRS representing our native coordinate system.

as_cartopy_projection()
Return a cartopy projection representing our native map.

This will be the same as the *as_cartopy_crs()* for map projections but for spherical coord systems (which are not map projections) we use a map projection, such as PlateCarree.

xml_element(*doc*, *attrs=None*)
Default behaviour for coord systems.

ellipsoid
Ellipsoid definition (*GeogCS* or None).

false_easting
X offset from planar origin in metres.

false_northing
Y offset from planar origin in metres.

grid_mapping_name = 'albers_conical_equal_area'

latitude_of_projection_origin
True latitude of planar origin in degrees.

longitude_of_central_meridian
True longitude of planar central meridian in degrees.

standard_parallels
The one or two latitudes of correct scale (tuple of 1 or 2 floats).

Abstract base class for coordinate systems.

class iris.coord_systems.**CoordSystem**
Abstract base class for coordinate systems.

abstract as_cartopy_crs()
Return a cartopy CRS representing our native coordinate system.

abstract as_cartopy_projection()
Return a cartopy projection representing our native map.

This will be the same as the *as_cartopy_crs()* for map projections but for spherical coord systems (which are not map projections) we use a map projection, such as PlateCarree.

xml_element(*doc*, *attrs=None*)
Default behaviour for coord systems.

grid_mapping_name = None

A geographic (ellipsoidal) coordinate system, defined by the shape of the Earth and a prime meridian.

class iris.coord_systems.**GeogCS**(*semi_major_axis=None*, *semi_minor_axis=None*, *inverse_flattening=None*, *longitude_of_prime_meridian=None*)

Creates a new GeogCS.

Kwargs:

- **semi_major_axis, semi_minor_axis:** Axes of ellipsoid, in metres. At least one must be given (see note below).
- **inverse_flattening:** Can be omitted if both axes given (see note below). Defaults to 0.0 .
- **longitude_of_prime_meridian:** Specifies the prime meridian on the ellipsoid, in degrees. Defaults to 0.0 .

If just semi_major_axis is set, with no semi_minor_axis or inverse_flattening, then a perfect sphere is created from the given radius.

If just two of semi_major_axis, semi_minor_axis, and inverse_flattening are given the missing element is calculated from the formula: $flattening = (major - minor)/major$

Currently, Iris will not allow over-specification (all three ellipsoid parameters).

Examples:

```
cs = GeogCS(6371229)
pp_cs = GeogCS(iris.fileformats.pp.EARTH_RADIUS)
airy1830 = GeogCS(semi_major_axis=6377563.396,
                  semi_minor_axis=6356256.909)
airy1830 = GeogCS(semi_major_axis=6377563.396,
                  inverse_flattening=299.3249646)
custom_cs = GeogCS(6400000, 6300000)
```

as_cartopy_crs()
Return a cartopy CRS representing our native coordinate system.

as_cartopy_globe()

as_cartopy_projection()
Return a cartopy projection representing our native map.

This will be the same as the *as_cartopy_crs()* for map projections but for spherical coord systems (which are not map projections) we use a map projection, such as PlateCarree.

xml_element(*doc*)
Default behaviour for coord systems.

grid_mapping_name = 'latitude_longitude'

inverse_flattening
$1/f$ where $f = (a - b)/a$.

longitude_of_prime_meridian
Describes 'zero' on the ellipsoid in degrees.

semi_major_axis
Major radius of the ellipsoid in metres.

semi_minor_axis
Minor radius of the ellipsoid in metres.

A geostationary satellite image map projection.

class iris.coord_systems.**Geostationary**(*latitude_of_projection_origin*, *longitude_of_projection_origin*, *perspective_point_height*, *sweep_angle_axis*, *false_easting=None*, *false_northing=None*, *ellipsoid=None*)

Constructs a Geostationary coord system.

Args:

- **latitude_of_projection_origin:** True latitude of planar origin in degrees.
- **longitude_of_projection_origin:** True longitude of planar origin in degrees.
- **perspective_point_height:** Altitude of satellite in metres above the surface of the ellipsoid.
- **sweep_angle_axis (string):** The axis along which the satellite instrument sweeps - 'x' or 'y'.

Kwargs:

- **false_easting:** X offset from planar origin in metres. Defaults to 0.0 .
- **false_northing:** Y offset from planar origin in metres. Defaults to 0.0 .
- **ellipsoid (*GeogCS*):** If given, defines the ellipsoid.

as_cartopy_crs()
Return a cartopy CRS representing our native coordinate system.

as_cartopy_projection()
Return a cartopy projection representing our native map.

This will be the same as the *as_cartopy_crs()* for map projections but for spherical coord systems (which are not map projections) we use a map projection, such as PlateCarree.

xml_element(*doc*, *attrs=None*)
Default behaviour for coord systems.

ellipsoid
Ellipsoid definition (*GeogCS* or None).

false_easting
X offset from planar origin in metres.

false_northing
Y offset from planar origin in metres.

grid_mapping_name = 'geostationary'

latitude_of_projection_origin
True latitude of planar origin in degrees.

longitude_of_projection_origin
True longitude of planar origin in degrees.

perspective_point_height
Altitude of satellite in metres.

sweep_angle_axis
The sweep angle axis (string 'x' or 'y').

A coordinate system in the Lambert Azimuthal Equal Area projection.

class iris.coord_systems.**LambertAzimuthalEqualArea**(*latitude_of_projection_origin=None*, *longitude_of_projection_origin=None*, *false_easting=None*, *false_northing=None*, *ellipsoid=None*)

Constructs a Lambert Azimuthal Equal Area coord system.

Kwargs:

- **latitude_of_projection_origin:** True latitude of planar origin in degrees. Defaults to 0.0 .
- **longitude_of_projection_origin:** True longitude of planar origin in degrees. Defaults to 0.0 .
- **false_easting:** X offset from planar origin in metres. Defaults to 0.0 .
- **false_northing:** Y offset from planar origin in metres. Defaults to 0.0 .
- **ellipsoid (*GeogCS*):** If given, defines the ellipsoid.

as_cartopy_crs()
Return a cartopy CRS representing our native coordinate system.

as_cartopy_projection()
Return a cartopy projection representing our native map.

This will be the same as the *as_cartopy_crs()* for map projections but for spherical coord systems (which are not map projections) we use a map projection, such as PlateCarree.

xml_element(*doc*, *attrs=None*)
Default behaviour for coord systems.

ellipsoid
Ellipsoid definition (*GeogCS* or None).

false_easting
X offset from planar origin in metres.

false_northing
Y offset from planar origin in metres.

grid_mapping_name = 'lambert_azimuthal_equal_area'

latitude_of_projection_origin
True latitude of planar origin in degrees.

longitude_of_projection_origin
True longitude of planar origin in degrees.

A coordinate system in the Lambert Conformal conic projection.

class iris.coord_systems.**LambertConformal**(*central_lat=None*, *central_lon=None*, *false_easting=None*, *false_northing=None*, *secant_latitudes=None*, *ellipsoid=None*)

Constructs a LambertConformal coord system.

Kwargs:

- **central_lat:** The latitude of "unitary scale". Defaults to 39.0 .
- **central_lon:** The central longitude. Defaults to -96.0 .
- **false_easting:** X offset from planar origin in metres. Defaults to 0.0 .
- **false_northing:** Y offset from planar origin in metres. Defaults to 0.0 .
- **secant_latitudes (number or iterable of 1 or 2 numbers):** Latitudes of secant intersection. One or two. Defaults to (33.0, 45.0).
- **ellipsoid (*GeogCS*):** If given, defines the ellipsoid.

as_cartopy_crs()
Return a cartopy CRS representing our native coordinate system.

as_cartopy_projection()
Return a cartopy projection representing our native map.

This will be the same as the *as_cartopy_crs()* for map projections but for spherical coord systems (which are not map projections) we use a map projection, such as PlateCarree.

xml_element(*doc*, *attrs=None*)
Default behaviour for coord systems.

central_lat
True latitude of planar origin in degrees.

central_lon
True longitude of planar origin in degrees.

ellipsoid
Ellipsoid definition (*GeogCS* or None).

false_easting
X offset from planar origin in metres.

false_northing
Y offset from planar origin in metres.

grid_mapping_name = 'lambert_conformal_conic'

secant_latitudes
The standard parallels of the cone (tuple of 1 or 2 floats).

A coordinate system in the Mercator projection.

class iris.coord_systems.**Mercator**(*longitude_of_projection_origin=None*, *ellipsoid=None*, *standard_parallel=None*)

Constructs a Mercator coord system.

Kwargs:

- **longitude_of_projection_origin:** True longitude of planar origin in degrees. Defaults to 0.0 .
- **ellipsoid (*GeogCS*):** If given, defines the ellipsoid.
- **standard_parallel:** The latitude where the scale is 1. Defaults to 0.0 .

as_cartopy_crs()
Return a cartopy CRS representing our native coordinate system.

as_cartopy_projection()
Return a cartopy projection representing our native map.

This will be the same as the *as_cartopy_crs()* for map projections but for spherical coord systems (which are not map projections) we use a map projection, such as PlateCarree.

xml_element(*doc*, *attrs=None*)
Default behaviour for coord systems.

ellipsoid
Ellipsoid definition (*GeogCS* or None).

grid_mapping_name = 'mercator'

longitude_of_projection_origin
True longitude of planar origin in degrees.

standard_parallel
The latitude where the scale is 1.

A Specific transverse mercator projection on a specific ellipsoid.

class iris.coord_systems.**OSGB**
A Specific transverse mercator projection on a specific ellipsoid.

as_cartopy_crs()
Return a cartopy CRS representing our native coordinate system.

as_cartopy_projection()
Return a cartopy projection representing our native map.

This will be the same as the *as_cartopy_crs()* for map projections but for spherical coord systems (which are not map projections) we use a map projection, such as PlateCarree.

xml_element(*doc*, *attrs=None*)
Default behaviour for coord systems.

ellipsoid
Ellipsoid definition (*GeogCS* or None).

false_easting
X offset from planar origin in metres.

false_northing
Y offset from planar origin in metres.

grid_mapping_name = 'transverse_mercator'

latitude_of_projection_origin
True latitude of planar origin in degrees.

longitude_of_central_meridian
True longitude of planar origin in degrees.

scale_factor_at_central_meridian
Scale factor at the centre longitude.

An orthographic map projection.

class iris.coord_systems.**Orthographic**(*latitude_of_projection_origin*, *longitude_of_projection_origin*, *false_easting=None*, *false_northing=None*, *ellipsoid=None*)

Constructs an Orthographic coord system.

Args:

- **latitude_of_projection_origin:** True latitude of planar origin in degrees.
- **longitude_of_projection_origin:** True longitude of planar origin in degrees.

Kwargs:

- **false_easting:** X offset from planar origin in metres. Defaults to 0.0 .
- **false_northing:** Y offset from planar origin in metres. Defaults to 0.0 .
- **ellipsoid (*GeogCS*):** If given, defines the ellipsoid.

as_cartopy_crs()
Return a cartopy CRS representing our native coordinate system.

as_cartopy_projection()
Return a cartopy projection representing our native map.

This will be the same as the *as_cartopy_crs()* for map projections but for spherical coord systems (which are not map projections) we use a map projection, such as PlateCarree.

xml_element(*doc*, *attrs=None*)
Default behaviour for coord systems.

ellipsoid
Ellipsoid definition (*GeogCS* or None).

false_easting
X offset from planar origin in metres.

false_northing
Y offset from planar origin in metres.

grid_mapping_name = 'orthographic'

latitude_of_projection_origin
True latitude of planar origin in degrees.

longitude_of_projection_origin
True longitude of planar origin in degrees.

A coordinate system with rotated pole, on an optional *GeogCS*.

class iris.coord_systems.**RotatedGeogCS**(*grid_north_pole_latitude*, *grid_north_pole_longitude*, *north_pole_grid_longitude=None*, *ellipsoid=None*)

Constructs a coordinate system with rotated pole, on an optional *GeogCS*.

Args:

- **grid_north_pole_latitude:** The true latitude of the rotated pole in degrees.
- **grid_north_pole_longitude:** The true longitude of the rotated pole in degrees.

Kwargs:

- **north_pole_grid_longitude:** Longitude of true north pole in rotated grid, in degrees. Defaults to 0.0 .
- **ellipsoid (*GeogCS*):** If given, defines the ellipsoid.

Examples:

```
rotated_cs = RotatedGeogCS(30, 30)
another_cs = RotatedGeogCS(30, 30,
                           ellipsoid=GeogCS(6400000, 6300000))
```

as_cartopy_crs()
Return a cartopy CRS representing our native coordinate system.

as_cartopy_projection()
Return a cartopy projection representing our native map.

This will be the same as the *as_cartopy_crs()* for map projections but for spherical coord systems (which are not map projections) we use a map projection, such as PlateCarree.

xml_element(*doc*)
Default behaviour for coord systems.

ellipsoid
Ellipsoid definition (*GeogCS* or None).

grid_mapping_name = 'rotated_latitude_longitude'

grid_north_pole_latitude
The true latitude of the rotated pole in degrees.

grid_north_pole_longitude
The true longitude of the rotated pole in degrees.

north_pole_grid_longitude
Longitude of true north pole in rotated grid in degrees.

A stereographic map projection.

class iris.coord_systems.**Stereographic**(*central_lat*, *central_lon*, *false_easting=None*, *false_northing=None*, *true_scale_lat=None*, *ellipsoid=None*)

Constructs a Stereographic coord system.

Args:

- **central_lat:** The latitude of the pole.
- **central_lon:** The central longitude, which aligns with the y axis.

Kwargs:

- **false_easting:** X offset from planar origin in metres. Defaults to 0.0 .
- **false_northing:** Y offset from planar origin in metres. Defaults to 0.0 .
- **true_scale_lat:** Latitude of true scale.
- **ellipsoid (*GeogCS*):** If given, defines the ellipsoid.

as_cartopy_crs()
Return a cartopy CRS representing our native coordinate system.

as_cartopy_projection()
Return a cartopy projection representing our native map.

This will be the same as the *as_cartopy_crs()* for map projections but for spherical coord systems (which are not map projections) we use a map projection, such as PlateCarree.

xml_element(*doc*, *attrs=None*)
Default behaviour for coord systems.

central_lat
True latitude of planar origin in degrees.

central_lon
True longitude of planar origin in degrees.

ellipsoid
Ellipsoid definition (*GeogCS* or None).

false_easting
X offset from planar origin in metres.

false_northing
Y offset from planar origin in metres.

grid_mapping_name = 'stereographic'

true_scale_lat
Latitude of true scale.

A cylindrical map projection, with XY coordinates measured in metres.

class iris.coord_systems.**TransverseMercator**(*latitude_of_projection_origin*, *longitude_of_central_meridian*, *false_easting=None*, *false_northing=None*, *scale_factor_at_central_meridian=None*, *ellipsoid=None*)

Constructs a TransverseMercator object.

Args:

- **latitude_of_projection_origin:** True latitude of planar origin in degrees.
- **longitude_of_central_meridian:** True longitude of planar origin in degrees.

Kwargs:

- **false_easting:** X offset from planar origin in metres. Defaults to 0.0 .
- **false_northing:** Y offset from planar origin in metres. Defaults to 0.0 .
- **scale_factor_at_central_meridian:** Reduces the cylinder to slice through the ellipsoid (secant form). Used to provide TWO longitudes of zero distortion in the area of interest. Defaults to 1.0 .
- **ellipsoid (*GeogCS*):** If given, defines the ellipsoid.

Example:

```
airy1830 = GeogCS(6377563.396, 6356256.909)
osgb = TransverseMercator(49, -2, 400000, -100000, 0.9996012717,
                          ellipsoid=airy1830)
```

as_cartopy_crs()
Return a cartopy CRS representing our native coordinate system.

as_cartopy_projection()
Return a cartopy projection representing our native map.

This will be the same as the *as_cartopy_crs()* for map projections but for spherical coord systems (which are not map projections) we use a map projection, such as PlateCarree.

xml_element(*doc*, *attrs=None*)
Default behaviour for coord systems.

ellipsoid
Ellipsoid definition (*GeogCS* or None).

false_easting
X offset from planar origin in metres.

false_northing
Y offset from planar origin in metres.

grid_mapping_name = 'transverse_mercator'

latitude_of_projection_origin
True latitude of planar origin in degrees.

longitude_of_central_meridian
True longitude of planar origin in degrees.

scale_factor_at_central_meridian
Scale factor at the centre longitude.

A vertical/near-side perspective satellite image map projection.

class iris.coord_systems.**VerticalPerspective**(*latitude_of_projection_origin*, *longitude_of_projection_origin*, *perspective_point_height*, *false_easting=None*, *false_northing=None*, *ellipsoid=None*)

Constructs a Vertical Perspective coord system.

Args:

- **latitude_of_projection_origin:** True latitude of planar origin in degrees.

- **longitude_of_projection_origin:** True longitude of planar origin in degrees.
- **perspective_point_height:** Altitude of satellite in metres above the surface of the ellipsoid.

Kwargs:

- **false_easting:** X offset from planar origin in metres. Defaults to 0.0 .
- **false_northing:** Y offset from planar origin in metres. Defaults to 0.0 .
- **ellipsoid (*GeogCS*):** If given, defines the ellipsoid.

as_cartopy_crs()
Return a cartopy CRS representing our native coordinate system.

as_cartopy_projection()
Return a cartopy projection representing our native map.

This will be the same as the *as_cartopy_crs()* for map projections but for spherical coord systems (which are not map projections) we use a map projection, such as PlateCarree.

xml_element(*doc*, *attrs=None*)
Default behaviour for coord systems.

ellipsoid
Ellipsoid definition (*GeogCS* or None).

false_easting
X offset from planar origin in metres.

false_northing
Y offset from planar origin in metres.

grid_mapping_name = 'vertical_perspective'

latitude_of_projection_origin
True latitude of planar origin in degrees.

longitude_of_projection_origin
True longitude of planar origin in degrees.

perspective_point_height
Altitude of satellite in metres.

27.7 iris.coords

Definitions of coordinates and other dimensional metadata.

In this module:

- *AncillaryVariable*
- *AuxCoord*
- *Cell*
- *CellMeasure*
- *CellMethod*
- *Coord*

- *CoordExtent*
- *DimCoord*

Superclass for dimensional metadata.

class iris.coords.**AncillaryVariable**(*data*, *standard_name=None*, *long_name=None*, *var_name=None*, *units=None*, *attributes=None*)

Constructs a single ancillary variable.

Args:

- **data:** The values of the ancillary variable.

Kwargs:

- **standard_name:** CF standard name of the ancillary variable.
- **long_name:** Descriptive name of the ancillary variable.
- **var_name:** The netCDF variable name for the ancillary variable.
- **units** The `Unit` of the ancillary variable's values. Can be a string, which will be converted to a Unit object.
- **attributes** A dictionary containing other cf and user-defined attributes.

__binary_operator__(*other*, *mode_constant*)

Common code which is called by add, sub, mul and div

Mode constant is one of ADD, SUB, MUL, DIV, RDIV

Note: The unit is *not* changed when doing scalar operations on a metadata object. This means that a metadata object which represents "10 meters" when multiplied by a scalar i.e. "1000" would result in a metadata object of "10000 meters". An alternative approach could be taken to multiply the *unit* by 1000 and the resultant metadata object would represent "10 kilometers".

__getitem__(*keys*)

Returns a new dimensional metadata whose values are obtained by conventional array indexing.

Note: Indexing of a circular coordinate results in a non-circular coordinate if the overall shape of the coordinate changes after indexing.

---

convert_units(*unit*)

Change the units, converting the values of the metadata.

copy(*values=None*)

Returns a copy of this dimensional metadata object.

Kwargs:

- **values** An array of values for the new dimensional metadata object. This may be a different shape to the original values array being copied.

core_data()

The data array at the core of this ancillary variable, which may be a NumPy array or a dask array.

cube_dims(*cube*)
Return the cube dimensions of this AncillaryVariable.

Equivalent to "cube.ancillary_variable_dims(self)".

has_bounds()
Return a boolean indicating whether the current dimensional metadata object has a bounds array.

has_lazy_data()
Return a boolean indicating whether the ancillary variable's data array is a lazy dask array or not.

is_compatible(*other*, *ignore=None*)
Return whether the current dimensional metadata object is compatible with another.

lazy_data()
Return a lazy array representing the ancillary variable's data.

Accessing this method will never cause the data values to be loaded. Similarly, calling methods on, or indexing, the returned Array will not cause the ancillary variable to have loaded data.

If the data have already been loaded for the ancillary variable, the returned Array will be a new lazy array wrapper.

Returns A lazy array, representing the ancillary variable data array.

name(*default=None*, *token=False*)
Returns a string name representing the identity of the metadata.

First it tries standard name, then it tries the long name, then the NetCDF variable name, before falling-back to a default value, which itself defaults to the string 'unknown'.

Kwargs:

- **default:** The fall-back string representing the default name. Defaults to the string 'unknown'.
- **token:** If True, ensures that the name returned satisfies the criteria for the characters required by a valid NetCDF name. If it is not possible to return a valid name, then a ValueError exception is raised. Defaults to False.

Returns String.

rename(*name*)
Changes the human-readable name.

If 'name' is a valid standard name it will assign it to *standard_name*, otherwise it will assign it to *long_name*.

xml_element(*doc*)
Create the `xml.dom.minidom.Element` that describes this `_DimensionalMetadata`.

Args:

- **doc:** The parent `xml.dom.minidom.Document`.

Returns The `xml.dom.minidom.Element` that will describe this `_DimensionalMetadata`, and the dictionary of attributes that require to be added to this element.

property attributes

property data

property dtype
The NumPy dtype of the current dimensional metadata object, as specified by its values.

property long_name
The CF Metadata long name for the object.

property metadata

property ndim
Return the number of dimensions of the current dimensional metadata object.

property shape
The fundamental shape of the metadata, expressed as a tuple.

property standard_name
The CF Metadata standard name for the object.

property units
The S.I. unit of the object.

property var_name
The NetCDF variable name for the object.

A CF auxiliary coordinate.

class iris.coords.**AuxCoord**(**args*, ***kwargs*)
Create a coordinate with **mutable** points and bounds.

Args:

- **points:** The values (or value in the case of a scalar coordinate) for each cell of the coordinate.

Kwargs:

- **standard_name:** CF standard name of the coordinate.
- **long_name:** Descriptive name of the coordinate.
- **var_name:** The netCDF variable name for the coordinate.
- **units** The `Unit` of the coordinate's values. Can be a string, which will be converted to a Unit object.
- **bounds** An array of values describing the bounds of each cell. Given n bounds for each cell, the shape of the bounds array should be points.shape + (n,). For example, a 1D coordinate with 100 points and two bounds per cell would have a bounds array of shape (100, 2) Note if the data is a climatology, *climatological* should be set.
- **attributes** A dictionary containing other CF and user-defined attributes.
- **coord_system** A *CoordSystem* representing the coordinate system of the coordinate, e.g., a *GeogCS* for a longitude coordinate.
- **climatological (bool):** When True: the coordinate is a NetCDF climatological time axis. When True: saving in NetCDF will give the coordinate variable a 'climatology' attribute and will create a boundary variable called '<coordinate-name>_climatology' in place of a standard bounds attribute and bounds variable. Will set to True when a climatological time axis is loaded from NetCDF. Always False if no bounds exist.

__binary_operator__(*other*, *mode_constant*)
Common code which is called by add, sub, mul and div

Mode constant is one of ADD, SUB, MUL, DIV, RDIV

Note: The unit is *not* changed when doing scalar operations on a metadata object. This means that a metadata object which represents "10 meters" when multiplied by a scalar i.e. "1000" would result in a metadata object of "10000 meters". An alternative approach could be taken to multiply the *unit* by 1000 and the resultant metadata object would represent "10 kilometers".

__getitem__(*keys*)
Returns a new dimensional metadata whose values are obtained by conventional array indexing.

Note: Indexing of a circular coordinate results in a non-circular coordinate if the overall shape of the coordinate changes after indexing.

---

cell(*index*)
Return the single *Cell* instance which results from slicing the points/bounds with the given index.

cells()
Returns an iterable of Cell instances for this Coord.

For example:

```
for cell in self.cells():
    ...
```

collapsed(*dims_to_collapse=None*)
Returns a copy of this coordinate, which has been collapsed along the specified dimensions.

Replaces the points & bounds with a simple bounded region.

contiguous_bounds()
Returns the N+1 bound values for a contiguous bounded 1D coordinate of length N, or the (N+1, M+1) bound values for a contiguous bounded 2D coordinate of shape (N, M).

Only 1D or 2D coordinates are supported.

Note: If the coordinate has bounds, this method assumes they are contiguous.

If the coordinate is 1D and does not have bounds, this method will return bounds positioned halfway between the coordinate's points.

If the coordinate is 2D and does not have bounds, an error will be raised.

---

convert_units(*unit*)
Change the coordinate's units, converting the values in its points and bounds arrays.

For example, if a coordinate's *units* attribute is set to radians then:

```
coord.convert_units('degrees')
```

will change the coordinate's *units* attribute to degrees and multiply each value in *points* and *bounds* by 180.0/π.

copy(*points=None*, *bounds=None*)
Returns a copy of this coordinate.

Kwargs:

- **points: A points array for the new coordinate.** This may be a different shape to the points of the coordinate being copied.
- **bounds: A bounds array for the new coordinate.** Given n bounds for each cell, the shape of the bounds array should be points.shape + (n,). For example, a 1d coordinate with 100 points and two bounds per cell would have a bounds array of shape (100, 2).

Note: If the points argument is specified and bounds are not, the resulting coordinate will have no bounds.

---

core_bounds()
The points array at the core of this coord, which may be a NumPy array or a dask array.

core_points()
The points array at the core of this coord, which may be a NumPy array or a dask array.

cube_dims(*cube*)
Return the cube dimensions of this Coord.

Equivalent to "cube.coord_dims(self)".

classmethod from_coord(*coord*)
Create a new Coord of this type, from the given coordinate.

guess_bounds(*bound_position=0.5*)
Add contiguous bounds to a coordinate, calculated from its points.

Puts a cell boundary at the specified fraction between each point and the next, plus extrapolated lowermost and uppermost bound points, so that each point lies within a cell.

With regularly spaced points, the resulting bounds will also be regular, and all points lie at the same position within their cell. With irregular points, the first and last cells are given the same widths as the ones next to them.

Kwargs:

- **bound_position:** The desired position of the bounds relative to the position of the points.

Note: An error is raised if the coordinate already has bounds, is not one-dimensional, or is not monotonic.

Note: Unevenly spaced values, such from a wrapped longitude range, can produce unexpected results : In such cases you should assign suitable values directly to the

bounds property, instead.

---

has_bounds()
Return a boolean indicating whether the coord has a bounds array.

has_lazy_bounds()
Return a boolean indicating whether the coord's bounds array is a lazy dask array or not.

has_lazy_points()
Return a boolean indicating whether the coord's points array is a lazy dask array or not.

intersect(*other*, *return_indices=False*)
Returns a new coordinate from the intersection of two coordinates.

Both coordinates must be compatible as defined by *is_compatible()*.

Kwargs:
- **return_indices:** If True, changes the return behaviour to return the intersection indices for the "self" coordinate.

is_compatible(*other*, *ignore=None*)
Return whether the coordinate is compatible with another.

Compatibility is determined by comparing *iris.coords.Coord.name()*, *iris.coords.Coord.units*, *iris.coords.Coord.coord_system* and *iris.coords.Coord.attributes* that are present in both objects.

Args:
- **other:** An instance of *iris.coords.Coord*, `iris.common.CoordMetadata` or `iris.common.DimCoordMetadata`.
- **ignore:** A single attribute key or iterable of attribute keys to ignore when comparing the coordinates. Default is None. To ignore all attributes, set this to other.attributes.

Returns Boolean.

is_contiguous(*rtol=1e-05*, *atol=1e-08*)
Return True if, and only if, this Coord is bounded with contiguous bounds to within the specified relative and absolute tolerances.

1D coords are contiguous if the upper bound of a cell aligns, within a tolerance, to the lower bound of the next cell along.

2D coords, with 4 bounds, are contiguous if the lower right corner of each cell aligns with the lower left corner of the cell to the right of it, and the upper left corner of each cell aligns with the lower left corner of the cell above it.

Args:
- **rtol:** The relative tolerance parameter (default is 1e-05).
- **atol:** The absolute tolerance parameter (default is 1e-08).

Returns Boolean.

is_monotonic()
Return True if, and only if, this Coord is monotonic.

lazy_bounds()
Return a lazy array representing the coord bounds.

Accessing this method will never cause the bounds values to be loaded. Similarly, calling methods on, or indexing, the returned Array will not cause the coord to have loaded bounds.

If the data have already been loaded for the coord, the returned Array will be a new lazy array wrapper.

Returns A lazy array representing the coord bounds array or *None* if the coord does not have bounds.

lazy_points()

Return a lazy array representing the coord points.

Accessing this method will never cause the points values to be loaded. Similarly, calling methods on, or indexing, the returned Array will not cause the coord to have loaded points.

If the data have already been loaded for the coord, the returned Array will be a new lazy array wrapper.

Returns A lazy array, representing the coord points array.

name(*default=None*, *token=False*)

Returns a string name representing the identity of the metadata.

First it tries standard name, then it tries the long name, then the NetCDF variable name, before falling-back to a default value, which itself defaults to the string 'unknown'.

Kwargs:

- **default:** The fall-back string representing the default name. Defaults to the string 'unknown'.
- **token:** If True, ensures that the name returned satisfies the criteria for the characters required by a valid NetCDF name. If it is not possible to return a valid name, then a ValueError exception is raised. Defaults to False.

Returns String.

nearest_neighbour_index(*point*)

Returns the index of the cell nearest to the given point.

Only works for one-dimensional coordinates.

For example:

```
>>> cube = iris.load_cube(iris.sample_data_path('ostia_
↪monthly.nc'))
>>> cube.coord('latitude').nearest_neighbour_index(0)
9
>>> cube.coord('longitude').nearest_neighbour_index(10)
12
```

Note: If the coordinate contains bounds, these will be used to determine the nearest neighbour instead of the point values.

Note: For circular coordinates, the 'nearest' point can wrap around to the other end of the values.

rename(*name*)
Changes the human-readable name.

If 'name' is a valid standard name it will assign it to *standard_name*, otherwise it will assign it to *long_name*.

xml_element(*doc*)
Create the `xml.dom.minidom.Element` that describes this *Coord*.

Args:
- **doc:** The parent `xml.dom.minidom.Document`.

Returns The `xml.dom.minidom.Element` that will describe this *DimCoord*, and the dictionary of attributes that require to be added to this element.

property attributes

property bounds
The coordinate bounds values, as a NumPy array, or None if no bound values are defined.

Note: The shape of the bound array should be: `points.shape + (n_bounds, )`.

---

property bounds_dtype
The NumPy dtype of the coord's bounds. Will be *None* if the coord does not have bounds.

property climatological
A boolean that controls whether the coordinate is a climatological time axis, in which case the bounds represent a climatological period rather than a normal period.

Always reads as False if there are no bounds. On set, the input value is cast to a boolean, exceptions raised if units are not time units or if there are no bounds.

property coord_system
The coordinate-system of the coordinate.

property dtype
The NumPy dtype of the current dimensional metadata object, as specified by its values.

property long_name
The CF Metadata long name for the object.

property metadata

property nbounds
Return the number of bounds that this coordinate has (0 for no bounds).

property ndim
Return the number of dimensions of the current dimensional metadata object.

property points
The coordinate points values as a NumPy array.

property shape
The fundamental shape of the metadata, expressed as a tuple.

property standard_name
The CF Metadata standard name for the object.

property units
The S.I. unit of the object.

property var_name
The NetCDF variable name for the object.

An immutable representation of a single cell of a coordinate, including the sample point and/or boundary position.

Notes on cell comparison:

Cells are compared in two ways, depending on whether they are compared to another Cell, or to a number/string.

Cell-Cell comparison is defined to produce a strict ordering. If two cells are not exactly equal (i.e. including whether they both define bounds or not) then they will have a consistent relative order.

Cell-number and Cell-string comparison is defined to support Constraint matching. The number/string will equal the Cell if, and only if, it is within the Cell (including on the boundary). The relative comparisons (lt, le, ..) are defined to be consistent with this interpretation. So for a given value *n* and Cell *cell*, only one of the following can be true:

n < cell
n == cell
n > cell

Similarly, *n* <= *cell* implies either *n* < *cell* or *n* == *cell*. And *n* >= *cell* implies either *n* > *cell* or *n* == *cell*.

class iris.coords.**Cell**(*point=None*, *bound=None*)
Construct a Cell from point or point-and-bound information.

__common_cmp__(*other*, *operator_method*)
Common method called by the rich comparison operators. The method of checking equality depends on the type of the object to be compared.

Cell vs Cell comparison is used to define a strict order. Non-Cell vs Cell comparison is used to define Constraint matching.

__eq__(*other*)
Compares Cell equality depending on the type of the object to be compared.

static __new__(*cls*, *point=None*, *bound=None*)
Construct a Cell from point or point-and-bound information.

contains_point(*point*)
For a bounded cell, returns whether the given point lies within the bounds.

Note: The test carried out is equivalent to min(bound) <= point <= max(bound).

---

count(*value*, */*)
Return number of occurrences of value.

index(*value*, *start=0*, *stop=9223372036854775807*, */*)
Return first index of value.

Raises ValueError if the value is not present.

bound
Alias for field number 1

point
Alias for field number 0

A CF Cell Measure, providing area or volume properties of a cell where these cannot be inferred from the Coordinates and Coordinate Reference System.

class iris.coords.**CellMeasure**(*data*, *standard_name=None*, *long_name=None*, *var_name=None*, *units=None*, *attributes=None*, *measure=None*)
Constructs a single cell measure.

Args:

- **data:** The values of the measure for each cell. Either a 'real' array (numpy.ndarray) or a 'lazy' array (dask.array.Array).

Kwargs:

- **standard_name:** CF standard name of the coordinate.
- **long_name:** Descriptive name of the coordinate.
- **var_name:** The netCDF variable name for the coordinate.
- **units** The Unit of the coordinate's values. Can be a string, which will be converted to a Unit object.
- **attributes** A dictionary containing other CF and user-defined attributes.
- **measure** A string describing the type of measure. Supported values are 'area' and 'volume'. The default is 'area'.

__binary_operator__(*other*, *mode_constant*)
Common code which is called by add, sub, mul and div

Mode constant is one of ADD, SUB, MUL, DIV, RDIV

Note: The unit is *not* changed when doing scalar operations on a metadata object. This means that a metadata object which represents "10 meters" when multiplied by a scalar i.e. "1000" would result in a metadata object of "10000 meters". An alternative approach could be taken to multiply the *unit* by 1000 and the resultant metadata object would represent "10 kilometers".

__getitem__(*keys*)
Returns a new dimensional metadata whose values are obtained by conventional array indexing.

Note: Indexing of a circular coordinate results in a non-circular coordinate if the overall shape of the coordinate changes after indexing.

---

convert_units(*unit*)
Change the units, converting the values of the metadata.

copy(*values=None*)
Returns a copy of this dimensional metadata object.

Kwargs:
- **values** An array of values for the new dimensional metadata object. This may be a different shape to the original values array being copied.

core_data()
The data array at the core of this ancillary variable, which may be a NumPy array or a dask array.

cube_dims(*cube*)
Return the cube dimensions of this CellMeasure.

Equivalent to "cube.cell_measure_dims(self)".

has_bounds()
Return a boolean indicating whether the current dimensional metadata object has a bounds array.

has_lazy_data()
Return a boolean indicating whether the ancillary variable's data array is a lazy dask array or not.

is_compatible(*other*, *ignore=None*)
Return whether the current dimensional metadata object is compatible with another.

lazy_data()
Return a lazy array representing the ancillary variable's data.

Accessing this method will never cause the data values to be loaded. Similarly, calling methods on, or indexing, the returned Array will not cause the ancillary variable to have loaded data.

If the data have already been loaded for the ancillary variable, the returned Array will be a new lazy array wrapper.

Returns A lazy array, representing the ancillary variable data array.

name(*default=None*, *token=False*)
Returns a string name representing the identity of the metadata.

First it tries standard name, then it tries the long name, then the NetCDF variable name, before falling-back to a default value, which itself defaults to the string 'unknown'.

Kwargs:
- **default:** The fall-back string representing the default name. Defaults to the string 'unknown'.
- **token:** If True, ensures that the name returned satisfies the criteria for the characters required by a valid NetCDF name. If it is not possible to return a valid name, then a ValueError exception is raised. Defaults to False.

Returns String.

rename(*name*)
Changes the human-readable name.

If 'name' is a valid standard name it will assign it to *standard_name*, otherwise it will assign it to *long_name*.

xml_element(*doc*)
Create the `xml.dom.minidom.Element` that describes this *CellMeasure*.

Args:
- **doc:** The parent `xml.dom.minidom.Document`.

Returns The `xml.dom.minidom.Element` that describes this *CellMeasure*.

property attributes

property data

property dtype
The NumPy dtype of the current dimensional metadata object, as specified by its values.

property long_name
The CF Metadata long name for the object.

property measure
String naming the measure type.

property metadata

property ndim
Return the number of dimensions of the current dimensional metadata object.

property shape
The fundamental shape of the metadata, expressed as a tuple.

property standard_name
The CF Metadata standard name for the object.

property units
The S.I. unit of the object.

property var_name
The NetCDF variable name for the object.

Represents a sub-cell pre-processing operation.

class `iris.coords.`**CellMethod**(*method*, *coords=None*, *intervals=None*, *comments=None*)

Args:
- **method:** The name of the operation.

Kwargs:
- **coords:** A single instance or sequence of *Coord* instances or coordinate names.
- **intervals:** A single string, or a sequence strings, describing the intervals within the cell method.
- **comments:** A single string, or a sequence strings, containing any additional comments.

__str__()
Return a custom string representation of CellMethod

xml_element(*doc*)
Create the `xml.dom.minidom.Element` that describes this *CellMethod*.

Args:
- **doc:** The parent `xml.dom.minidom.Document`.

Returns The `xml.dom.minidom.Element` that describes this *CellMethod*.

comments = None
Additional comments.

coord_names = None
The tuple of coordinate names over which the operation was applied.

intervals = None
A description of the original intervals over which the operation was applied.

method = None
The name of the operation that was applied. e.g. "mean", "max", etc.

Abstract base class for coordinates.

class iris.coords.**Coord**(*points*, *standard_name=None*, *long_name=None*, *var_name=None*, *units=None*, *bounds=None*, *attributes=None*, *coord_system=None*, *climatological=False*)
Coordinate abstract base class. As of `v3.0.0` you **cannot** create an instance of *Coord*.

Args:

- **points:** The values (or value in the case of a scalar coordinate) for each cell of the coordinate.

Kwargs:

- **standard_name:** CF standard name of the coordinate.
- **long_name:** Descriptive name of the coordinate.
- **var_name:** The netCDF variable name for the coordinate.
- **units** The `Unit` of the coordinate's values. Can be a string, which will be converted to a Unit object.
- **bounds** An array of values describing the bounds of each cell. Given n bounds for each cell, the shape of the bounds array should be points.shape + (n,). For example, a 1D coordinate with 100 points and two bounds per cell would have a bounds array of shape (100, 2) Note if the data is a climatology, *climatological* should be set.
- **attributes** A dictionary containing other CF and user-defined attributes.
- **coord_system** A *CoordSystem* representing the coordinate system of the coordinate, e.g., a *GeogCS* for a longitude coordinate.
- **climatological (bool):** When True: the coordinate is a NetCDF climatological time axis. When True: saving in NetCDF will give the coordinate variable a 'climatology' attribute and will create a boundary variable called '<coordinate-name>_climatology' in place of a standard bounds attribute and bounds variable. Will set to True when a climatological time axis is loaded from NetCDF. Always False if no bounds exist.

__binary_operator__(*other*, *mode_constant*)
Common code which is called by add, sub, mul and div

Mode constant is one of ADD, SUB, MUL, DIV, RDIV

Note: The unit is *not* changed when doing scalar operations on a metadata object. This means that a metadata object which represents "10 meters" when multiplied by a scalar i.e. "1000" would result in a metadata object of "10000 meters". An alternative approach could be taken to multiply the *unit* by 1000 and the resultant metadata object would represent "10 kilometers".

__getitem__(*keys*)
Returns a new dimensional metadata whose values are obtained by conventional array indexing.

Note: Indexing of a circular coordinate results in a non-circular coordinate if the overall shape of the coordinate changes after indexing.

---

cell(*index*)
Return the single *Cell* instance which results from slicing the points/bounds with the given index.

cells()
Returns an iterable of Cell instances for this Coord.

For example:

```
for cell in self.cells():
    ...
```

collapsed(*dims_to_collapse=None*)
Returns a copy of this coordinate, which has been collapsed along the specified dimensions.

Replaces the points & bounds with a simple bounded region.

contiguous_bounds()
Returns the N+1 bound values for a contiguous bounded 1D coordinate of length N, or the (N+1, M+1) bound values for a contiguous bounded 2D coordinate of shape (N, M).

Only 1D or 2D coordinates are supported.

Note: If the coordinate has bounds, this method assumes they are contiguous.

If the coordinate is 1D and does not have bounds, this method will return bounds positioned halfway between the coordinate's points.

If the coordinate is 2D and does not have bounds, an error will be raised.

---

convert_units(*unit*)
Change the coordinate's units, converting the values in its points and bounds arrays.

For example, if a coordinate's *units* attribute is set to radians then:

```
coord.convert_units('degrees')
```

will change the coordinate's *units* attribute to degrees and multiply each value in *points* and *bounds* by 180.0/π.

copy(*points=None*, *bounds=None*)
Returns a copy of this coordinate.

Kwargs:

- **points: A points array for the new coordinate.** This may be a different shape to the points of the coordinate being copied.
- **bounds: A bounds array for the new coordinate.** Given n bounds for each cell, the shape of the bounds array should be points.shape + (n,). For example, a 1d coordinate with 100 points and two bounds per cell would have a bounds array of shape (100, 2).

Note: If the points argument is specified and bounds are not, the resulting coordinate will have no bounds.

---

core_bounds()
The points array at the core of this coord, which may be a NumPy array or a dask array.

core_points()
The points array at the core of this coord, which may be a NumPy array or a dask array.

cube_dims(*cube*)
Return the cube dimensions of this Coord.

Equivalent to "cube.coord_dims(self)".

classmethod from_coord(*coord*)
Create a new Coord of this type, from the given coordinate.

guess_bounds(*bound_position=0.5*)
Add contiguous bounds to a coordinate, calculated from its points.

Puts a cell boundary at the specified fraction between each point and the next, plus extrapolated lowermost and uppermost bound points, so that each point lies within a cell.

With regularly spaced points, the resulting bounds will also be regular, and all points lie at the same position within their cell. With irregular points, the first and last cells are given the same widths as the ones next to them.

Kwargs:

- **bound_position:** The desired position of the bounds relative to the position of the points.

Note: An error is raised if the coordinate already has bounds, is not one-dimensional, or is not monotonic.

Note: Unevenly spaced values, such from a wrapped longitude range, can produce unexpected results : In such cases you should assign suitable values directly to the

bounds property, instead.

---

has_bounds()
Return a boolean indicating whether the coord has a bounds array.

has_lazy_bounds()
Return a boolean indicating whether the coord's bounds array is a lazy dask array or not.

has_lazy_points()
Return a boolean indicating whether the coord's points array is a lazy dask array or not.

intersect(*other*, *return_indices=False*)
Returns a new coordinate from the intersection of two coordinates.

Both coordinates must be compatible as defined by *is_compatible()*.

Kwargs:
- **return_indices:** If True, changes the return behaviour to return the intersection indices for the "self" coordinate.

is_compatible(*other*, *ignore=None*)
Return whether the coordinate is compatible with another.

Compatibility is determined by comparing *iris.coords.Coord.name()*, *iris.coords.Coord.units*, *iris.coords.Coord.coord_system* and *iris.coords.Coord.attributes* that are present in both objects.

Args:
- **other:** An instance of *iris.coords.Coord*, `iris.common.CoordMetadata` or `iris.common.DimCoordMetadata`.
- **ignore:** A single attribute key or iterable of attribute keys to ignore when comparing the coordinates. Default is None. To ignore all attributes, set this to other.attributes.

Returns Boolean.

is_contiguous(*rtol=1e-05*, *atol=1e-08*)
Return True if, and only if, this Coord is bounded with contiguous bounds to within the specified relative and absolute tolerances.

1D coords are contiguous if the upper bound of a cell aligns, within a tolerance, to the lower bound of the next cell along.

2D coords, with 4 bounds, are contiguous if the lower right corner of each cell aligns with the lower left corner of the cell to the right of it, and the upper left corner of each cell aligns with the lower left corner of the cell above it.

Args:
- **rtol:** The relative tolerance parameter (default is 1e-05).
- **atol:** The absolute tolerance parameter (default is 1e-08).

Returns Boolean.

is_monotonic()
Return True if, and only if, this Coord is monotonic.

lazy_bounds()
Return a lazy array representing the coord bounds.

Accessing this method will never cause the bounds values to be loaded. Similarly, calling methods on, or indexing, the returned Array will not cause the coord to have loaded bounds.

If the data have already been loaded for the coord, the returned Array will be a new lazy array wrapper.

Returns A lazy array representing the coord bounds array or *None* if the coord does not have bounds.

lazy_points()

Return a lazy array representing the coord points.

Accessing this method will never cause the points values to be loaded. Similarly, calling methods on, or indexing, the returned Array will not cause the coord to have loaded points.

If the data have already been loaded for the coord, the returned Array will be a new lazy array wrapper.

Returns A lazy array, representing the coord points array.

name(*default=None*, *token=False*)

Returns a string name representing the identity of the metadata.

First it tries standard name, then it tries the long name, then the NetCDF variable name, before falling-back to a default value, which itself defaults to the string 'unknown'.

Kwargs:

- **default:** The fall-back string representing the default name. Defaults to the string 'unknown'.
- **token:** If True, ensures that the name returned satisfies the criteria for the characters required by a valid NetCDF name. If it is not possible to return a valid name, then a ValueError exception is raised. Defaults to False.

Returns String.

nearest_neighbour_index(*point*)

Returns the index of the cell nearest to the given point.

Only works for one-dimensional coordinates.

For example:

```
>>> cube = iris.load_cube(iris.sample_data_path('ostia_
↪monthly.nc'))
>>> cube.coord('latitude').nearest_neighbour_index(0)
9
>>> cube.coord('longitude').nearest_neighbour_index(10)
12
```

Note: If the coordinate contains bounds, these will be used to determine the nearest neighbour instead of the point values.

Note: For circular coordinates, the 'nearest' point can wrap around to the other end of the values.

rename(*name*)
Changes the human-readable name.

If 'name' is a valid standard name it will assign it to *standard_name*, otherwise it will assign it to *long_name*.

xml_element(*doc*)
Create the `xml.dom.minidom.Element` that describes this *Coord*.

Args:
- **doc:** The parent `xml.dom.minidom.Document`.

Returns The `xml.dom.minidom.Element` that will describe this *DimCoord*, and the dictionary of attributes that require to be added to this element.

property attributes

property bounds
The coordinate bounds values, as a NumPy array, or None if no bound values are defined.

Note: The shape of the bound array should be: `points.shape + (n_bounds, )`.

---

property bounds_dtype
The NumPy dtype of the coord's bounds. Will be *None* if the coord does not have bounds.

property climatological
A boolean that controls whether the coordinate is a climatological time axis, in which case the bounds represent a climatological period rather than a normal period.

Always reads as False if there are no bounds. On set, the input value is cast to a boolean, exceptions raised if units are not time units or if there are no bounds.

property coord_system
Relevant coordinate system (if any).

property dtype
The NumPy dtype of the current dimensional metadata object, as specified by its values.

property long_name
The CF Metadata long name for the object.

property metadata

property nbounds
Return the number of bounds that this coordinate has (0 for no bounds).

property ndim
Return the number of dimensions of the current dimensional metadata object.

property points
The coordinate points values as a NumPy array.

property shape
The fundamental shape of the metadata, expressed as a tuple.

property standard_name
The CF Metadata standard name for the object.

property units
The S.I. unit of the object.

property var_name
The NetCDF variable name for the object.

Defines a range of values for a coordinate.

class iris.coords.**CoordExtent**(*name_or_coord*, *minimum*, *maximum*, *min_inclusive=True*, *max_inclusive=True*)
Create a CoordExtent for the specified coordinate and range of values.

Args:

- **name_or_coord** Either a coordinate name or a coordinate, as defined in *iris.cube.Cube.coords()*.
- **minimum** The minimum value of the range to select.
- **maximum** The maximum value of the range to select.

Kwargs:

- **min_inclusive** If True, coordinate values equal to *minimum* will be included in the selection. Default is True.
- **max_inclusive** If True, coordinate values equal to *maximum* will be included in the selection. Default is True.

static __new__(*cls*, *name_or_coord*, *minimum*, *maximum*, *min_inclusive=True*, *max_inclusive=True*)
Create a CoordExtent for the specified coordinate and range of values.

Args:
- **name_or_coord** Either a coordinate name or a coordinate, as defined in *iris.cube.Cube.coords()*.
- **minimum** The minimum value of the range to select.
- **maximum** The maximum value of the range to select.

Kwargs:
- **min_inclusive** If True, coordinate values equal to *minimum* will be included in the selection. Default is True.
- **max_inclusive** If True, coordinate values equal to *maximum* will be included in the selection. Default is True.

count(*value*, */*)
Return number of occurrences of value.

index(*value*, *start=0*, *stop=9223372036854775807*, */*)
Return first index of value.

Raises ValueError if the value is not present.

max_inclusive
Alias for field number 4

maximum
Alias for field number 2

min_inclusive
Alias for field number 3

minimum
Alias for field number 1

name_or_coord
Alias for field number 0

A coordinate that is 1D, and numeric, with values that have a strict monotonic ordering. Missing values are not permitted in a *DimCoord*.

class iris.coords.**DimCoord**(*points*, *standard_name=None*, *long_name=None*, *var_name=None*, *units=None*, *bounds=None*, *attributes=None*, *coord_system=None*, *circular=False*, *climatological=False*)

Create a 1D, numeric, and strictly monotonic coordinate with **immutable** points and bounds.

Missing values are not permitted.

Args:

- **points:** 1D numpy array-like of values (or single value in the case of a scalar coordinate) for each cell of the coordinate. The values must be strictly monotonic and masked values are not allowed.

Kwargs:

- **standard_name:** CF standard name of the coordinate.
- **long_name:** Descriptive name of the coordinate.
- **var_name:** The netCDF variable name for the coordinate.
- **units:** The `Unit` of the coordinate's values. Can be a string, which will be converted to a Unit object.
- **bounds:** An array of values describing the bounds of each cell. Given n bounds and m cells, the shape of the bounds array should be (m, n). For each bound, the values must be strictly monotonic along the cells, and the direction of monotonicity must be consistent across the bounds. For example, a DimCoord with 100 points and two bounds per cell would have a bounds array of shape (100, 2), and the slices `bounds[:, 0]` and `bounds[:, 1]` would be monotonic in the same direction. Masked values are not allowed. Note if the data is a climatology, *climatological* should be set.
- **attributes:** A dictionary containing other CF and user-defined attributes.
- **coord_system:** A *CoordSystem* representing the coordinate system of the coordinate, e.g., a *GeogCS* for a longitude coordinate.
- **circular (bool):** Whether the coordinate wraps by the `modulus` i.e., the longitude coordinate wraps around the full great circle.
- **climatological (bool):** When True: the coordinate is a NetCDF climatological time axis. When True: saving in NetCDF will give the coordinate variable a 'climatology' attribute and will create a boundary variable called '<coordinate-name>_climatology' in place of a standard bounds attribute and bounds variable. Will set to True when a climatological time axis is loaded from NetCDF. Always False if no bounds exist.

__binary_operator__(*other*, *mode_constant*)
Common code which is called by add, sub, mul and div

Mode constant is one of ADD, SUB, MUL, DIV, RDIV

Note: The unit is *not* changed when doing scalar operations on a metadata object. This means that a metadata object which represents "10 meters" when multiplied by a scalar i.e. "1000" would result in a metadata object of "10000 meters". An alternative approach could be taken to multiply the *unit* by 1000 and the resultant metadata object would represent "10 kilometers".

__deepcopy__() → Deep copy of coordinate.
Used if copy.deepcopy is called on a coordinate.

cell(*index*)
Return the single *Cell* instance which results from slicing the points/bounds with the given index.

cells()
Returns an iterable of Cell instances for this Coord.

For example:

```
for cell in self.cells():
    ...
```

collapsed(*dims_to_collapse=None*)
Returns a copy of this coordinate, which has been collapsed along the specified dimensions.

Replaces the points & bounds with a simple bounded region.

contiguous_bounds()
Returns the N+1 bound values for a contiguous bounded 1D coordinate of length N, or the (N+1, M+1) bound values for a contiguous bounded 2D coordinate of shape (N, M).

Only 1D or 2D coordinates are supported.

Note: If the coordinate has bounds, this method assumes they are contiguous.

If the coordinate is 1D and does not have bounds, this method will return bounds positioned halfway between the coordinate's points.

If the coordinate is 2D and does not have bounds, an error will be raised.

---

convert_units(*unit*)
Change the coordinate's units, converting the values in its points and bounds arrays.

For example, if a coordinate's *units* attribute is set to radians then:

```
coord.convert_units('degrees')
```

will change the coordinate's *units* attribute to degrees and multiply each value in *points* and *bounds* by $180.0/\pi$.

copy(*points=None*, *bounds=None*)
Returns a copy of this coordinate.

Kwargs:

- **points: A points array for the new coordinate.** This may be a different shape to the points of the coordinate being copied.
- **bounds: A bounds array for the new coordinate.** Given n bounds for each cell, the shape of the bounds array should be points.shape + (n,). For example, a 1d coordinate with 100 points and two bounds per cell would have a bounds array of shape (100, 2).

Note: If the points argument is specified and bounds are not, the resulting coordinate will have no bounds.

`core_bounds()`
The points array at the core of this coord, which may be a NumPy array or a dask array.

`core_points()`
The points array at the core of this coord, which may be a NumPy array or a dask array.

`cube_dims`(*cube*)
Return the cube dimensions of this Coord.

Equivalent to "cube.coord_dims(self)".

`classmethod from_coord`(*coord*)
Create a new Coord of this type, from the given coordinate.

`classmethod from_regular`(*zeroth*, *step*, *count*, *standard_name=None*, *long_name=None*, *var_name=None*, *units=None*, *attributes=None*, *coord_system=None*, *circular=False*, *climatological=False*, *with_bounds=False*)
Create a *DimCoord* with regularly spaced points, and optionally bounds.

The majority of the arguments are defined as for *Coord*, but those which differ are defined below.

Args:

- **zeroth:** The value *prior* to the first point value.
- **step:** The numeric difference between successive point values.
- **count:** The number of point values.

Kwargs:

- **with_bounds:** If True, the resulting DimCoord will possess bound values which are equally spaced around the points. Otherwise no bounds values will be defined. Defaults to False.

`guess_bounds`(*bound_position=0.5*)
Add contiguous bounds to a coordinate, calculated from its points.

Puts a cell boundary at the specified fraction between each point and the next, plus extrapolated lowermost and uppermost bound points, so that each point lies within a cell.

With regularly spaced points, the resulting bounds will also be regular, and all points lie at the same position within their cell. With irregular points, the first and last cells are given the same widths as the ones next to them.

Kwargs:

- **bound_position:** The desired position of the bounds relative to the position of the points.

Note: An error is raised if the coordinate already has bounds, is not one-dimensional, or is not monotonic.

Note: Unevenly spaced values, such from a wrapped longitude range, can produce unexpected results : In such cases you should assign suitable values directly to the bounds property, instead.

`has_bounds()`
Return a boolean indicating whether the coord has a bounds array.

`has_lazy_bounds()`
Return a boolean indicating whether the coord's bounds array is a lazy dask array or not.

`has_lazy_points()`
Return a boolean indicating whether the coord's points array is a lazy dask array or not.

`intersect(other, return_indices=False)`
Returns a new coordinate from the intersection of two coordinates.

Both coordinates must be compatible as defined by *iris.coords.Coord.is_compatible()*.

Kwargs:
- **return_indices:** If True, changes the return behaviour to return the intersection indices for the "self" coordinate.

`is_compatible(other, ignore=None)`
Return whether the coordinate is compatible with another.

Compatibility is determined by comparing *iris.coords.Coord.name()*, *iris.coords.Coord.units*, *iris.coords.Coord.coord_system* and *iris.coords.Coord.attributes* that are present in both objects.

Args:
- **other:** An instance of *iris.coords.Coord*, `iris.common.CoordMetadata` or `iris.common.DimCoordMetadata`.
- **ignore:** A single attribute key or iterable of attribute keys to ignore when comparing the coordinates. Default is None. To ignore all attributes, set this to other.attributes.

Returns Boolean.

`is_contiguous(rtol=1e-05, atol=1e-08)`
Return True if, and only if, this Coord is bounded with contiguous bounds to within the specified relative and absolute tolerances.

1D coords are contiguous if the upper bound of a cell aligns, within a tolerance, to the lower bound of the next cell along.

2D coords, with 4 bounds, are contiguous if the lower right corner of each cell aligns with the lower left corner of the cell to the right of it, and the upper left corner of each cell aligns with the lower left corner of the cell above it.

Args:

- **rtol:** The relative tolerance parameter (default is 1e-05).
- **atol:** The absolute tolerance parameter (default is 1e-08).

Returns Boolean.

is_monotonic()
Return True if, and only if, this Coord is monotonic.

lazy_bounds()
Return a lazy array representing the coord bounds.

Accessing this method will never cause the bounds values to be loaded. Similarly, calling methods on, or indexing, the returned Array will not cause the coord to have loaded bounds.

If the data have already been loaded for the coord, the returned Array will be a new lazy array wrapper.

Returns A lazy array representing the coord bounds array or *None* if the coord does not have bounds.

lazy_points()
Return a lazy array representing the coord points.

Accessing this method will never cause the points values to be loaded. Similarly, calling methods on, or indexing, the returned Array will not cause the coord to have loaded points.

If the data have already been loaded for the coord, the returned Array will be a new lazy array wrapper.

Returns A lazy array, representing the coord points array.

name(*default=None*, *token=False*)
Returns a string name representing the identity of the metadata.

First it tries standard name, then it tries the long name, then the NetCDF variable name, before falling-back to a default value, which itself defaults to the string 'unknown'.

Kwargs:

- **default:** The fall-back string representing the default name. Defaults to the string 'unknown'.
- **token:** If True, ensures that the name returned satisfies the criteria for the characters required by a valid NetCDF name. If it is not possible to return a valid name, then a ValueError exception is raised. Defaults to False.

Returns String.

nearest_neighbour_index(*point*)
Returns the index of the cell nearest to the given point.

Only works for one-dimensional coordinates.

For example:

```
>>> cube = iris.load_cube(iris.sample_data_path('ostia_
↪monthly.nc'))
>>> cube.coord('latitude').nearest_neighbour_index(0)
9
>>> cube.coord('longitude').nearest_neighbour_index(10)
12
```

Note: If the coordinate contains bounds, these will be used to determine the nearest neighbour instead of the point values.

Note: For circular coordinates, the 'nearest' point can wrap around to the other end of the values.

rename(*name*)
Changes the human-readable name.

If 'name' is a valid standard name it will assign it to *standard_name*, otherwise it will assign it to *long_name*.

xml_element(*doc*)
Create the `xml.dom.minidom.Element` that describes this *DimCoord*.

Args:
- **doc:** The parent `xml.dom.minidom.Document`.

Returns The `xml.dom.minidom.Element` that describes this *DimCoord*.

property attributes

property bounds
The coordinate bounds values, as a NumPy array, or None if no bound values are defined.

Note: The shape of the bound array should be: `points.shape + (n_bounds, )`.

property bounds_dtype
The NumPy dtype of the coord's bounds. Will be *None* if the coord does not have bounds.

property circular
Whether the coordinate wraps by `coord.units.modulus`.

property climatological
A boolean that controls whether the coordinate is a climatological time axis, in which case the bounds represent a climatological period rather than a normal period.

Always reads as False if there are no bounds. On set, the input value is cast to a boolean, exceptions raised if units are not time units or if there are no bounds.

property coord_system
The coordinate-system of the coordinate.

property dtype
The NumPy dtype of the current dimensional metadata object, as specified by its values.

property long_name
The CF Metadata long name for the object.

property metadata

property nbounds
Return the number of bounds that this coordinate has (0 for no bounds).

property ndim
Return the number of dimensions of the current dimensional metadata object.

property points
The coordinate points values as a NumPy array.

property shape
The fundamental shape of the metadata, expressed as a tuple.

property standard_name
The CF Metadata standard name for the object.

property units
The S.I. unit of the object.

property var_name
The NetCDF variable name for the object.

27.8 iris.cube

Classes for representing multi-dimensional data with metadata.

In this module:

- *Cube*
- *CubeList*

A single Iris cube of data and metadata.

Typically obtained from *iris.load()*, *iris.load_cube()*, *iris.load_cubes()*, or from the manipulation of existing cubes.

For example:

```
>>> cube = iris.load_cube(iris.sample_data_path('air_temp.pp'))
>>> print(cube)
air_temperature / (K)               (latitude: 73; longitude: 96)
    Dimension coordinates:
        latitude                           x              -
        longitude                          -              x
    Scalar coordinates:
        forecast_period             6477 hours, bound=(-28083.0, 6477.0) hours
        forecast_reference_time     1998-03-01 03:00:00
        pressure                    1000.0 hPa
        time                        1998-12-01 00:00:00, bound=(1994-12-01␣
↪00:00:00, 1998-12-01 00:00:00)
    Cell methods:
        mean within years           time
        mean over years             time
    Attributes:
        STASH                       m01s16i203
        source                      Data from Met Office Unified Model
```

See the *user guide* for more information.

class iris.cube.**Cube**(*data*, *standard_name=None*, *long_name=None*, *var_name=None*, *units=None*, *attributes=None*, *cell_methods=None*, *dim_coords_and_dims=None*, *aux_coords_and_dims=None*, *aux_factories=None*, *cell_measures_and_dims=None*, *ancillary_variables_and_dims=None*)

Creates a cube with data and optional metadata.

Not typically used - normally cubes are obtained by loading data (e.g. *iris.load()*) or from manipulating existing cubes.

Args:

- **data** This object defines the shape of the cube and the phenomenon value in each cell.

  `data` can be a dask array, a NumPy array, a NumPy array subclass (such as `numpy.ma.MaskedArray`), or array_like (as described in `numpy.asarray()`).

  See *Cube.data*.

Kwargs:

- **standard_name** The standard name for the Cube's data.
- **long_name** An unconstrained description of the cube.
- **var_name** The netCDF variable name for the cube.
- **units** The unit of the cube, e.g. `"m s-1"` or `"kelvin"`.
- **attributes** A dictionary of cube attributes
- **cell_methods** A tuple of CellMethod objects, generally set by Iris, e.g. `(CellMethod("mean", coords='latitude'), )`.
- **dim_coords_and_dims** A list of coordinates with scalar dimension mappings, e.g `[(lat_coord, 0), (lon_coord, 1)]`.
- **aux_coords_and_dims** A list of coordinates with dimension mappings, e.g `[(lat_coord, 0), (lon_coord, (0, 1))]`. See also *Cube.add_dim_coord()* and *Cube.add_aux_coord()*.
- **aux_factories** A list of auxiliary coordinate factories. See *iris.aux_factory*.
- **cell_measures_and_dims** A list of CellMeasures with dimension mappings.
- **ancillary_variables_and_dims** A list of AncillaryVariables with dimension mappings.

For example::

```
>>> from iris.coords import DimCoord
>>> from iris.cube import Cube
>>> latitude = DimCoord(np.linspace(-90, 90, 4),
...                     standard_name='latitude',
...                     units='degrees')
>>> longitude = DimCoord(np.linspace(45, 360, 8),
...                      standard_name='longitude',
...                      units='degrees')
>>> cube = Cube(np.zeros((4, 8), np.float32),
```

(continues on next page)

(continued from previous page)

```
...             dim_coords_and_dims=[(latitude, 0),
...                                  (longitude, 1)])
```

__copy__()
Shallow copying is disallowed for Cubes.

__getitem__(*keys*)
Cube indexing (through use of square bracket notation) has been implemented at the data level. That is, the indices provided to this method should be aligned to the data of the cube, and thus the indices requested must be applicable directly to the cube.data attribute. All metadata will be subsequently indexed appropriately.

add_ancillary_variable(*ancillary_variable*, *data_dims=None*)
Adds a CF ancillary variable to the cube.

Args:
- **ancillary_variable** The *iris.coords.AncillaryVariable* instance to be added to the cube

Kwargs:
- **data_dims** Integer or iterable of integers giving the data dimensions spanned by the ancillary variable.

Raises a ValueError if an ancillary variable with identical metadata already exists on the cube.

add_aux_coord(*coord*, *data_dims=None*)
Adds a CF auxiliary coordinate to the cube.

Args:
- **coord** The *iris.coords.DimCoord* or *iris.coords.AuxCoord* instance to add to the cube.

Kwargs:
- **data_dims** Integer or iterable of integers giving the data dimensions spanned by the coordinate.

Raises a ValueError if a coordinate with identical metadata already exists on the cube.

See also *Cube.remove_coord()*.

add_aux_factory(*aux_factory*)
Adds an auxiliary coordinate factory to the cube.

Args:
- **aux_factory** The *iris.aux_factory.AuxCoordFactory* instance to add.

add_cell_measure(*cell_measure*, *data_dims=None*)
Adds a CF cell measure to the cube.

Args:
- **cell_measure** The *iris.coords.CellMeasure* instance to add to the cube.

Kwargs:
- **data_dims** Integer or iterable of integers giving the data dimensions spanned by the coordinate.

Raises a ValueError if a cell_measure with identical metadata already exists on the cube.

See also *Cube.remove_cell_measure()*.

add_cell_method(*cell_method*)
Add a *CellMethod* to the Cube.

add_dim_coord(*dim_coord*, *data_dim*)
Add a CF coordinate to the cube.

Args:
- **dim_coord** The *iris.coords.DimCoord* instance to add to the cube.
- **data_dim** Integer giving the data dimension spanned by the coordinate.

Raises a ValueError if a coordinate with identical metadata already exists on the cube or if a coord already exists for the given dimension.

See also *Cube.remove_coord()*.

aggregated_by(*coords*, *aggregator*, ***kwargs*)
Perform aggregation over the cube given one or more "group coordinates".

A "group coordinate" is a coordinate where repeating values represent a single group, such as a month coordinate on a daily time slice. Repeated values will form a group even if they are not consecutive.

The group coordinates must all be over the same cube dimension. Each common value group identified over all the group-by coordinates is collapsed using the provided aggregator.

Args:
- **coords (list of coord names or *iris.coords.Coord* instances):** One or more coordinates over which group aggregation is to be performed.
- **aggregator (*iris.analysis.Aggregator*):** Aggregator to be applied to each group.

Kwargs:
- **kwargs:** Aggregator and aggregation function keyword arguments.

Returns *iris.cube.Cube*.

For example:

```
>>> import iris
>>> import iris.analysis
>>> import iris.coord_categorisation as cat
>>> fname = iris.sample_data_path('ostia_monthly.nc')
>>> cube = iris.load_cube(fname, 'surface_temperature')
>>> cat.add_year(cube, 'time', name='year')
>>> new_cube = cube.aggregated_by('year', iris.analysis.MEAN)
>>> print(new_cube)
surface_temperature / (K)           (time: 5; latitude: 18;␣
↪longitude: 432)
    Dimension coordinates:
        time                             x            -       ␣
↪       -
        latitude                         -            x       ␣
↪       -
        longitude                        -            -       ␣
↪       x
    Auxiliary coordinates:
        forecast_reference_time          x            -       ␣
↪       -
        year                             x            -       ␣
↪       -                                                     (continues on next page)
```

(continued from previous page)

```
    Scalar coordinates:
        forecast_period             0 hours
    Cell methods:
        mean                        month, year
        mean                        year
    Attributes:
        Conventions                 CF-1.5
        STASH                       m01s00i024
```

ancillary_variable(*name_or_ancillary_variable=None*)

Return a single ancillary_variable given the same arguments as *Cube.ancillary_variables()*.

Note: If the arguments given do not result in precisely 1 ancillary_variable being matched, an *iris.exceptions.AncillaryVariableNotFoundError* is raised.

See also:

Cube.ancillary_variables() for full keyword documentation.

ancillary_variable_dims(*ancillary_variable*)

Returns a tuple of the data dimensions relevant to the given AncillaryVariable.

- **ancillary_variable (string or AncillaryVariable)** The (name of the) AncillaryVariable to look for.

ancillary_variables(*name_or_ancillary_variable=None*)

Return a list of ancillary variable in this cube fitting the given criteria.

Kwargs:

- **name_or_ancillary_variable** Either

  (a) a *standard_name*, *long_name*, or *var_name*. Defaults to value of *default* (which itself defaults to *unknown*) as defined in `iris.common.CFVariableMixin`.

  (b) a ancillary_variable instance with metadata equal to that of the desired ancillary_variables.

See also *Cube.ancillary_variable()*.

aux_factory(*name=None*, *standard_name=None*, *long_name=None*, *var_name=None*)

Returns the single coordinate factory that matches the criteria, or raises an error if not found.

Kwargs:

- **name** If not None, matches against factory.name().
- **standard_name** The CF standard name of the desired coordinate factory. If None, does not check for standard name.
- **long_name** An unconstrained description of the coordinate factory. If None, does not check for long_name.
- **var_name** The netCDF variable name of the desired coordinate factory. If None, does not check for var_name.

Note: If the arguments given do not result in precisely 1 coordinate factory being matched, an *iris.exceptions.CoordinateNotFoundError* is raised.

---

cell_measure(*name_or_cell_measure=None*)

Return a single cell_measure given the same arguments as *Cube.cell_measures()*.

Note: If the arguments given do not result in precisely 1 cell_measure being matched, an *iris.exceptions.CellMeasureNotFoundError* is raised.

See also:

Cube.cell_measures() for full keyword documentation.

cell_measure_dims(*cell_measure*)

Returns a tuple of the data dimensions relevant to the given CellMeasure.

- **cell_measure (string or CellMeasure)** The (name of the) cell measure to look for.

cell_measures(*name_or_cell_measure=None*)

Return a list of cell measures in this cube fitting the given criteria.

Kwargs:

- **name_or_cell_measure** Either

  (a) a *standard_name*, *long_name*, or *var_name*. Defaults to value of *default* (which itself defaults to *unknown*) as defined in `iris.common.CFVariableMixin`.

  (b) a cell_measure instance with metadata equal to that of the desired cell_measures.

See also *Cube.cell_measure()*.

collapsed(*coords*, *aggregator*, ***kwargs*)

Collapse one or more dimensions over the cube given the coordinate/s and an aggregation.

Examples of aggregations that may be used include *COUNT* and *MAX*.

Weighted aggregations (*iris.analysis.WeightedAggregator*) may also be supplied. These include *MEAN* and sum *SUM*.

Weighted aggregations support an optional *weights* keyword argument. If set, this should be supplied as an array of weights whose shape matches the cube. Values for latitude-longitude area weights may be calculated using *iris.analysis.cartography.area_weights()*.

Some Iris aggregators support "lazy" evaluation, meaning that cubes resulting from this method may represent data arrays which are not computed until the data is requested (e.g. via `cube.data` or `iris.save`). If lazy evaluation exists for the given aggregator it will be used wherever possible when this cube's data is itself a deferred array.

Args:

- **coords (string, coord or a list of strings/coords):** Coordinate names/coordinates over which the cube should be collapsed.
- **aggregator (*iris.analysis.Aggregator*):** Aggregator to be applied for collapse operation.

Kwargs:

- **kwargs:** Aggregation function keyword arguments.

Returns Collapsed cube.

For example:

```
>>> import iris
>>> import iris.analysis
>>> path = iris.sample_data_path('ostia_monthly.nc')
>>> cube = iris.load_cube(path)
>>> new_cube = cube.collapsed('longitude', iris.analysis.MEAN)
>>> print(new_cube)
surface_temperature / (K)           (time: 54; latitude: 18)
    Dimension coordinates:
        time                             x             -
        latitude                         -             x
    Auxiliary coordinates:
        forecast_reference_time          x             -
    Scalar coordinates:
        forecast_period                  0 hours
        longitude                        180.0 degrees, bound=(0.0,
↪ 360.0) degrees
    Cell methods:
        mean                             month, year
        mean                             longitude
    Attributes:
        Conventions                      CF-1.5
        STASH                            m01s00i024
```

Note: Some aggregations are not commutative and hence the order of processing is important i.e.:

```
tmp = cube.collapsed('realization', iris.analysis.VARIANCE)
result = tmp.collapsed('height', iris.analysis.VARIANCE)
```

is not necessarily the same result as:

```
tmp = cube.collapsed('height', iris.analysis.VARIANCE)
result2 = tmp.collapsed('realization', iris.analysis.VARIANCE)
```

Conversely operations which operate on more than one coordinate at the same time are commutative as they are combined internally into a single operation. Hence the order of the coordinates supplied in the list does not matter:

```
cube.collapsed(['longitude', 'latitude'],
               iris.analysis.VARIANCE)
```

is the same (apart from the logically equivalent cell methods that may be created etc.) as:

```
cube.collapsed(['latitude', 'longitude'],
               iris.analysis.VARIANCE)
```

---

convert_units(*unit*)

Change the cube's units, converting the values in the data array.

For example, if a cube's *units* are kelvin then:

```
cube.convert_units('celsius')
```

will change the cube's *units* attribute to celsius and subtract 273.15 from each value in *data*.

This operation preserves lazy data.

coord(*name_or_coord=None*, *standard_name=None*, *long_name=None*, *var_name=None*, *attributes=None*, *axis=None*, *contains_dimension=None*, *dimensions=None*, *coord_system=None*, *dim_coords=None*)
Return a single coord given the same arguments as *Cube.coords()*.

Note: If the arguments given do not result in precisely 1 coordinate being matched, an *iris.exceptions.CoordinateNotFoundError* is raised.

See also:

Cube.coords() for full keyword documentation.

coord_dims(*coord*)
Returns a tuple of the data dimensions relevant to the given coordinate.

When searching for the given coordinate in the cube the comparison is made using coordinate metadata equality. Hence the given coordinate instance need not exist on the cube, and may contain different coordinate values.

Args:

- **coord (string or coord)** The (name of the) coord to look for.

coord_system(*spec=None*)
Find the coordinate system of the given type.

If no target coordinate system is provided then find any available coordinate system.

Kwargs:

- **spec:** The the name or type of a coordinate system subclass. E.g.

```
cube.coord_system("GeogCS")
cube.coord_system(iris.coord_systems.GeogCS)
```

If spec is provided as a type it can be a superclass of any coordinate system found.

If spec is None, then find any available coordinate systems within the *iris.cube.Cube*.

Returns The *iris.coord_systems.CoordSystem* or None.

coords(*name_or_coord=None*, *standard_name=None*, *long_name=None*, *var_name=None*, *attributes=None*, *axis=None*, *contains_dimension=None*, *dimensions=None*, *coord_system=None*, *dim_coords=None*)
Return a list of coordinates in this cube fitting the given criteria.

Kwargs:

- **name_or_coord** Either

(a) a *standard_name*, *long_name*, or *var_name*. Defaults to value of *default* (which itself defaults to *unknown*) as defined in `iris.common.CFVariableMixin`.

(b) a coordinate instance with metadata equal to that of the desired coordinates. Accepts either a *iris.coords.DimCoord*, *iris.coords.AuxCoord*, *iris.aux_factory.AuxCoordFactory*, `iris.common.CoordMetadata` or `iris.common.DimCoordMetadata`.

- **standard_name** The CF standard name of the desired coordinate. If None, does not check for standard name.
- **long_name** An unconstrained description of the coordinate. If None, does not check for long_name.
- **var_name** The netCDF variable name of the desired coordinate. If None, does not check for var_name.
- **attributes** A dictionary of attributes desired on the coordinates. If None, does not check for attributes.
- **axis** The desired coordinate axis, see *iris.util.guess_coord_axis()*. If None, does not check for axis. Accepts the values 'X', 'Y', 'Z' and 'T' (case-insensitive).
- **contains_dimension** The desired coordinate contains the data dimension. If None, does not check for the dimension.
- **dimensions** The exact data dimensions of the desired coordinate. Coordinates with no data dimension can be found with an empty tuple or list (i.e. `()` or `[]`). If None, does not check for dimensions.
- **coord_system** Whether the desired coordinates have coordinate systems equal to the given coordinate system. If None, no check is done.
- **dim_coords** Set to True to only return coordinates that are the cube's dimension coordinates. Set to False to only return coordinates that are the cube's auxiliary and derived coordinates. If None, returns all coordinates.

See also *Cube.coord()*.

copy(*data=None*)

Returns a deep copy of this cube.

Kwargs:

- **data:** Replace the data of the cube copy with provided data payload.

Returns A copy instance of the *Cube*.

core_data()

Retrieve the data array of this *Cube* in its current state, which will either be real or lazy.

If this *Cube* has lazy data, accessing its data array via this method **will not** realise the data array. This means you can perform operations using this method that work equivalently on real or lazy data, and will maintain lazy data if present.

extract(*constraint*)

Filter the cube by the given constraint using *iris.Constraint.extract()* method.

has_lazy_data()

Details whether this *Cube* has lazy data.

Returns Boolean.

interpolate(*sample_points*, *scheme*, *collapse_scalar=True*)

Interpolate from this *Cube* to the given sample points using the given interpolation scheme.

Args:

- **sample_points:** A sequence of (coordinate, points) pairs over which to interpolate. The values for coordinates that correspond to dates or times may

optionally be supplied as datetime.datetime or cftime.datetime instances.

- **scheme:** The type of interpolation to use to interpolate from this *Cube* to the given sample points. The interpolation schemes currently available in Iris are:
  - *iris.analysis.Linear*, and
  - *iris.analysis.Nearest*.

Kwargs:

- **collapse_scalar:** Whether to collapse the dimension of scalar sample points in the resulting cube. Default is True.

Returns A cube interpolated at the given sample points. If *collapse_scalar* is True then the dimensionality of the cube will be the number of original cube dimensions minus the number of scalar coordinates.

For example:

```
>>> import datetime
>>> import iris
>>> path = iris.sample_data_path('uk_hires.pp')
>>> cube = iris.load_cube(path, 'air_potential_temperature')
>>> print(cube.summary(shorten=True))
air_potential_temperature / (K)     (time: 3; model_level_
↪number: 7; grid_latitude: 204; grid_longitude: 187)
>>> print(cube.coord('time'))
DimCoord([2009-11-19 10:00:00, 2009-11-19 11:00:00, 2009-11-
↪19 12:00:00], standard_name='time', calendar='gregorian')
>>> print(cube.coord('time').points)
[349618. 349619. 349620.]
>>> samples = [('time', 349618.5)]
>>> result = cube.interpolate(samples, iris.analysis.Linear())
>>> print(result.summary(shorten=True))
air_potential_temperature / (K)     (model_level_number: 7;␣
↪grid_latitude: 204; grid_longitude: 187)
>>> print(result.coord('time'))
DimCoord([2009-11-19 10:30:00], standard_name='time',␣
↪calendar='gregorian')
>>> print(result.coord('time').points)
[349618.5]
>>> # For datetime-like coordinates, we can also use
>>> # datetime-like objects.
>>> samples = [('time', datetime.datetime(2009, 11, 19, 10,␣
↪30))]
>>> result2 = cube.interpolate(samples, iris.analysis.
↪Linear())
>>> print(result2.summary(shorten=True))
air_potential_temperature / (K)     (model_level_number: 7;␣
↪grid_latitude: 204; grid_longitude: 187)
>>> print(result2.coord('time'))
DimCoord([2009-11-19 10:30:00], standard_name='time',␣
↪calendar='gregorian')
>>> print(result2.coord('time').points)
[349618.5]
>>> print(result == result2)
True
```

intersection(**args*, ***kwargs*)

Return the intersection of the cube with specified coordinate ranges.

Coordinate ranges can be specified as:

(a) instances of *iris.coords.CoordExtent*.

(b) keyword arguments, where the keyword name specifies the name of the coordinate (as defined in *iris.cube.Cube.coords()*) and the value defines the corresponding range of coordinate values as a tuple. The tuple must contain two, three, or four items corresponding to: (minimum, maximum, min_inclusive, max_inclusive). Where the items are defined as:

- **minimum** The minimum value of the range to select.
- **maximum** The maximum value of the range to select.
- **min_inclusive** If True, coordinate values equal to *minimum* will be included in the selection. Default is True.
- **max_inclusive** If True, coordinate values equal to *maximum* will be included in the selection. Default is True.

To perform an intersection that ignores any bounds on the coordinates, set the optional keyword argument *ignore_bounds* to True. Defaults to False.

Note: For ranges defined over "circular" coordinates (i.e. those where the *units* attribute has a modulus defined) the cube will be "rolled" to fit where necessary.

Warning: Currently this routine only works with "circular" coordinates (as defined in the previous note.)

For example:

```
>>> import iris
>>> cube = iris.load_cube(iris.sample_data_path('air_temp.pp
↪'))
>>> print(cube.coord('longitude').points[::10])
[   0.           37.49999237   74.99998474  112.49996948  149.
↪99996948
  187.49995422  224.99993896  262.49993896  299.99993896  337.
↪49990845]
>>> subset = cube.intersection(longitude=(30, 50))
>>> print(subset.coord('longitude').points)
[ 33.74999237  37.49999237  41.24998856  44.99998856  48.
↪74998856]
>>> subset = cube.intersection(longitude=(-10, 10))
>>> print(subset.coord('longitude').points)
[-7.50012207 -3.75012207  0.          3.75        7.5       ]
```

Returns A new *Cube* giving the subset of the cube which intersects with the requested coordinate intervals.

is_compatible(*other*, *ignore=None*)

Return whether the cube is compatible with another.

Compatibility is determined by comparing *iris.cube.Cube.name()*, *iris.cube.Cube.units*, *iris.cube.Cube.cell_methods* and *iris.cube.Cube.attributes* that are present in both objects.

Args:

- **other:** An instance of *iris.cube.Cube* or `iris.cube.CubeMetadata`.
- **ignore:** A single attribute key or iterable of attribute keys to ignore when comparing the cubes. Default is None. To ignore all attributes set this to other.attributes.

Returns Boolean.

See also:

iris.util.describe_diff()

Note: This function does not indicate whether the two cubes can be merged, instead it checks only the four items quoted above for equality. Determining whether two cubes will merge requires additional logic that is beyond the scope of this method.

`lazy_data`()

Return a "lazy array" representing the Cube data. A lazy array describes an array whose data values have not been loaded into memory from disk.

Accessing this method will never cause the Cube data to be loaded. Similarly, calling methods on, or indexing, the returned Array will not cause the Cube data to be loaded.

If the Cube data have already been loaded (for example by calling *data()*), the returned Array will be a view of the loaded cube data represented as a lazy array object. Note that this does _not_ make the Cube data lazy again; the Cube data remains loaded in memory.

Returns A lazy array, representing the Cube data.

`name`(*default=None*, *token=False*)

Returns a string name representing the identity of the metadata.

First it tries standard name, then it tries the long name, then the NetCDF variable name, before falling-back to a default value, which itself defaults to the string 'unknown'.

Kwargs:

- **default:** The fall-back string representing the default name. Defaults to the string 'unknown'.
- **token:** If True, ensures that the name returned satisfies the criteria for the characters required by a valid NetCDF name. If it is not possible to return a valid name, then a ValueError exception is raised. Defaults to False.

Returns String.

`regrid`(*grid*, *scheme*)

Regrid this *Cube* on to the given target *grid* using the given regridding *scheme*.

Args:

- **grid:** A *Cube* that defines the target grid.
- **scheme:** The type of regridding to use to regrid this cube onto the target grid. The regridding schemes in Iris currently include:
  - *iris.analysis.Linear**,
  - *iris.analysis.Nearest**,
  - *iris.analysis.AreaWeighted**,
  - *iris.analysis.UnstructuredNearest*,
  - *iris.analysis.PointInCell*,

* Supports lazy regridding.

Returns

A cube defined with the horizontal dimensions of the target grid and the other dimensions from this cube. The data values of this cube will be converted to values on the new grid according to the given regridding scheme.

The returned cube will have lazy data if the original cube has lazy data and the regridding scheme supports lazy regridding.

Note: Both the source and target cubes must have a CoordSystem, otherwise this function is not applicable.

---

remove_ancillary_variable(*ancillary_variable*)
Removes an ancillary variable from the cube.

Args:
- **ancillary_variable (string or AncillaryVariable)** The (name of the) AncillaryVariable to remove from the cube.

remove_aux_factory(*aux_factory*)
Removes the given auxiliary coordinate factory from the cube.

remove_cell_measure(*cell_measure*)
Removes a cell measure from the cube.

Args:
- **cell_measure (string or cell_measure)** The (name of the) cell measure to remove from the cube. As either

  (a) a *standard_name*, *long_name*, or *var_name*. Defaults to value of *default* (which itself defaults to *unknown*) as defined in `iris.common.CFVariableMixin`.

  (b) a cell_measure instance with metadata equal to that of the desired cell_measures.

Note: If the argument given does not represent a valid cell_measure on the cube, an *iris.exceptions.CellMeasureNotFoundError* is raised.

See also:

Cube.add_cell_measure()

remove_coord(*coord*)
Removes a coordinate from the cube.

Args:
- **coord (string or coord)** The (name of the) coordinate to remove from the cube.

See also *Cube.add_dim_coord()* and *Cube.add_aux_coord()*.

rename(*name*)
Changes the human-readable name.

If 'name' is a valid standard name it will assign it to *standard_name*, otherwise it will assign it to *long_name*.

replace_coord(*new_coord*)
Replace the coordinate whose metadata matches the given coordinate.

rolling_window(*coord*, *aggregator*, *window*, ***kwargs*)
Perform rolling window aggregation on a cube given a coordinate, an aggregation method and a window size.

Args:
- **coord (string/*iris.coords.Coord*):** The coordinate over which to perform the rolling window aggregation.
- **aggregator (*iris.analysis.Aggregator*):** Aggregator to be applied to the data.
- **window (int):** Size of window to use.

Kwargs:
- **kwargs:** Aggregator and aggregation function keyword arguments. The weights argument to the aggregator, if any, should be a 1d array with the same length as the chosen window.

Returns *iris.cube.Cube*.

Note: This operation does not yet have support for lazy evaluation.

For example:

```
>>> import iris, iris.analysis
>>> fname = iris.sample_data_path('GloSea4', 'ensemble_
↪010.pp')
>>> air_press = iris.load_cube(fname, 'surface_
↪temperature')
>>> print(air_press)
surface_temperature / (K)           (time: 6; latitude:␣
↪145; longitude: 192)
    Dimension coordinates:
        time                             x            -␣
↪               -
        latitude                         -            x␣
↪               -
        longitude                        -            -␣
↪               x
    Auxiliary coordinates:
        forecast_period                  x            -␣
↪               -
    Scalar coordinates:
        forecast_reference_time     2011-07-23 00:00:00
        realization                 10
    Cell methods:
        mean                        time (1 hour)
    Attributes:
        STASH                       m01s00i024
        source                      Data from Met␣
↪Office Unified Model
        um_version                  7.6
```

```
>>> print(air_press.rolling_window('time', iris.
↪analysis.MEAN, 3))
surface_temperature / (K)           (time: 4; latitude:␣
↪145; longitude: 192)
    Dimension coordinates:
        time                             x            -␣
↪               -
        latitude                         -            x␣
↪               -
        longitude                        -            -␣
↪               x
    Auxiliary coordinates:
        forecast_period                  x            -␣
↪               -
    Scalar coordinates:
        forecast_reference_time     2011-07-23 00:00:00
        realization                 10
    Cell methods:
        mean                        time (1 hour)
        mean                        time
    Attributes:
        STASH                       m01s00i024
        source                      Data from Met␣
↪Office Unified Model
        um_version                  7.6
```

Notice that the forecast_period dimension now represents the 4 possible windows of size 3 from the original cube.

slices(*ref_to_slice*, *ordered=True*)

Return an iterator of all subcubes given the coordinates or dimension indices desired to be present in each subcube.

Args:

- **ref_to_slice (string, coord, dimension index or a list of these):** Determines which dimensions will be returned in the subcubes (i.e. the dimensions that are not iterated over). A mix of input types can also be provided. They must all be orthogonal (i.e. point to different dimensions).

Kwargs:

- **ordered: if True, the order which the coords to slice or data_dims** are given will be the order in which they represent the data in the resulting cube slices. If False, the order will follow that of the source cube. Default is True.

Returns An iterator of subcubes.

For example, to get all 2d longitude/latitude subcubes from a multi-dimensional cube:

```
for sub_cube in cube.slices(['longitude', 'latitude']):
    print(sub_cube)
```

See also:

iris.cube.Cube.slices_over().

slices_over(*ref_to_slice*)

Return an iterator of all subcubes along a given coordinate or dimension index, or

multiple of these.

Args:

- **ref_to_slice (string, coord, dimension index or a list of these):** Determines which dimensions will be iterated along (i.e. the dimensions that are not returned in the subcubes). A mix of input types can also be provided.

Returns An iterator of subcubes.

For example, to get all subcubes along the time dimension:

```
for sub_cube in cube.slices_over('time'):
    print(sub_cube)
```

See also:

iris.cube.Cube.slices().

Note: The order of dimension references to slice along does not affect the order of returned items in the iterator; instead the ordering is based on the fastest-changing dimension.

---

subset(*coord*)

Get a subset of the cube by providing the desired resultant coordinate. If the coordinate provided applies to the whole cube; the whole cube is returned. As such, the operation is not strict.

summary(*shorten=False*, *name_padding=35*)

String summary of the Cube with name+units, a list of dim coord names versus length and, optionally, a summary of all other components.

Kwargs:

- **shorten (bool):** If set, produce a one-line summary of minimal width, showing only the cube name, units and dimensions. When not set (default), produces a full multi-line summary string.
- **name_padding (int):** Control the *minimum* width of the cube name + units, i.e. the indent of the dimension map section.

transpose(*new_order=None*)

Re-order the data dimensions of the cube in-place.

new_order - list of ints, optional By default, reverse the dimensions, otherwise permute the axes according to the values given.

Note: If defined, new_order must span all of the data dimensions.

Example usage:

```
# put the second dimension first, followed by the third␣
↪dimension,
and finally put the first dimension third::

    >>> cube.transpose([1, 2, 0])
```

xml(*checksum=False*, *order=True*, *byteorder=True*)

Returns a fully valid CubeML string representation of the Cube.

property attributes
A dictionary, with a few restricted keys, for arbitrary Cube metadata.

property aux_coords
Return a tuple of all the auxiliary coordinates, ordered by dimension(s).

property aux_factories
Return a tuple of all the coordinate factories.

property cell_methods
Tuple of *iris.coords.CellMethod* representing the processing done on the phenomenon.

property data
The numpy.ndarray representing the multi-dimensional data of the cube.

Note: Cubes obtained from netCDF, PP, and FieldsFile files will only populate this attribute on its first use.

To obtain the shape of the data without causing it to be loaded, use the Cube.shape attribute.

Example::

```
>>> fname = iris.sample_data_path('air_temp.pp')
>>> cube = iris.load_cube(fname, 'air_temperature')
>>> # cube.data does not yet have a value.
...
>>> print(cube.shape)
(73, 96)
>>> # cube.data still does not have a value.
...
>>> cube = cube[:10, :20]
>>> # cube.data still does not have a value.
...
>>> data = cube.data
>>> # Only now is the data loaded.
...
>>> print(data.shape)
(10, 20)
```

property derived_coords
Return a tuple of all the coordinates generated by the coordinate factories.

property dim_coords
Return a tuple of all the dimension coordinates, ordered by dimension.

Note: The length of the returned tuple is not necessarily the same as *Cube.ndim* as there may be dimensions on the cube without dimension coordinates. It is therefore unreliable to use the resulting tuple to identify the dimension coordinates for a given dimension - instead use the *Cube.coord()* method with the `dimensions` and `dim_coords` keyword arguments.

---

property dtype
The data type of the values in the data array of this *Cube*.

property long_name
The "long name" for the Cube's phenomenon.

property metadata

property ndim
The number of dimensions in the data of this cube.

property shape
The shape of the data of this cube.

property standard_name
The "standard name" for the Cube's phenomenon.

property units
An instance of `cf_units.Unit` describing the Cube's data.

property var_name
The netCDF variable name for the Cube.

All the functionality of a standard `list` with added "Cube" context.

class iris.cube.**CubeList**(*list_of_cubes=None*)
Given a `list` of cubes, return a CubeList instance.

__getitem__(*keys*)
x.__getitem__(y) <==> x[y]

__getslice__(*start*, *stop*)
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

static __new__(*cls*, *list_of_cubes=None*)
Given a `list` of cubes, return a CubeList instance.

__repr__()
Runs repr on every cube.

__str__()
Runs short *Cube.summary()* on every cube.

append(*object*, */*)
Append object to the end of the list.

clear()
Remove all items from list.

concatenate(*check_aux_coords=True*, *check_cell_measures=True*, *check_ancils=True*)
Concatenate the cubes over their common dimensions.

Kwargs:

- **check_aux_coords** Checks the auxiliary coordinates of the cubes match. This check is not applied to auxiliary coordinates that span the dimension the concatenation is occurring along. Defaults to True.
- **check_cell_measures** Checks the cell measures of the cubes match. This check is not applied to cell measures that span the dimension the concatenation is occurring along. Defaults to True.
- **check_ancils** Checks the ancillary variables of the cubes match. This check is not applied to ancillary variables that span the dimension the concatenation is occurring along. Defaults to True.

Returns A new *iris.cube.CubeList* of concatenated *iris.cube.Cube* instances.

This combines cubes with a common dimension coordinate, but occupying different regions of the coordinate value. The cubes are joined across that dimension.

For example:

```
>>> print(c1)
some_parameter / (unknown)          (y_vals: 2; x_vals: 4)
     Dimension coordinates:
          y_vals                             x          -
          x_vals                             -          x
>>> print(c1.coord('y_vals').points)
[4 5]
>>> print(c2)
some_parameter / (unknown)          (y_vals: 3; x_vals: 4)
     Dimension coordinates:
          y_vals                             x          -
          x_vals                             -          x
>>> print(c2.coord('y_vals').points)
[ 7  9 10]
>>> cube_list = iris.cube.CubeList([c1, c2])
>>> new_cube = cube_list.concatenate()[0]
>>> print(new_cube)
some_parameter / (unknown)          (y_vals: 5; x_vals: 4)
     Dimension coordinates:
          y_vals                             x          -
          x_vals                             -          x
>>> print(new_cube.coord('y_vals').points)
[ 4  5  7  9 10]
>>>
```

Contrast this with *iris.cube.CubeList.merge()*, which makes a new dimension from values of an auxiliary scalar coordinate.

Note: Cubes may contain 'extra' dimensional elements such as auxiliary coordinates, cell measures or ancillary variables. For a group of similar cubes to concatenate together into one output, all such elements which do not map to the concatenation axis must be identical in every input cube : these then appear unchanged in the output. Similarly, those elements which *do* map to the concatenation axis must have matching properties, but may have different data values : these then appear, concatenated, in the output cube. If any cubes in a group have dimensional elements which do not match correctly, the group will not concatenate to a single output cube.

Note: If time coordinates in the list of cubes have differing epochs then the cubes will not be able to be concatenated. If this occurs, use *iris.util.unify_time_units()* to normalise the epochs of the time coordinates so that the cubes can be concatenated.

Note: Concatenation cannot occur along an anonymous dimension.

concatenate_cube(*check_aux_coords=True*, *check_cell_measures=True*, *check_ancils=True*)

Return the concatenated contents of the *CubeList* as a single *Cube*.

If it is not possible to concatenate the *CubeList* into a single *Cube*, a *ConcatenateError* will be raised describing the reason for the failure.

Kwargs:

- **check_aux_coords** Checks the auxiliary coordinates of the cubes match. This check is not applied to auxiliary coordinates that span the dimension the concatenation is occurring along. Defaults to True.
- **check_cell_measures** Checks the cell measures of the cubes match. This check is not applied to cell measures that span the dimension the concatenation is occurring along. Defaults to True.
- **check_ancils** Checks the ancillary variables of the cubes match. This check is not applied to ancillary variables that span the dimension the concatenation is occurring along. Defaults to True.

Note: Concatenation cannot occur along an anonymous dimension.

copy()

Return a CubeList when CubeList.copy() is called.

count(*value*, */*)

Return number of occurrences of value.

extend(*iterable*, */*)

Extend list by appending elements from the iterable.

extract(*constraints*)

Filter each of the cubes which can be filtered by the given constraints.

This method iterates over each constraint given, and subsets each of the cubes in this CubeList where possible. Thus, a CubeList of length **n** when filtered with **m** constraints can generate a maximum of **m** * **n** cubes.

Args:

- **constraints (*Constraint* or iterable of constraints):** A single constraint or an iterable.

extract_cube(*constraint*)

Extract a single cube from a CubeList, and return it. Raise an error if the extract produces no cubes, or more than one.

Args:

- **constraint (*Constraint*):** The constraint to extract with.

extract_cubes(*constraints*)

Extract specific cubes from a CubeList, one for each given constraint. Each constraint must produce exactly one cube, otherwise an error is raised.

Args:

- **constraints (iterable of, or single, *Constraint*):** The constraints to extract with.

extract_overlapping(*coord_names*)

Returns a *CubeList* of cubes extracted over regions where the coordinates overlap, for the coordinates in coord_names.

Args:

- **coord_names:** A string or list of strings of the names of the coordinates over which to perform the extraction.

index(*value*, *start=0*, *stop=9223372036854775807*, */*)

Return first index of value.

Raises ValueError if the value is not present.

insert(*index*, *object*, */*)

Insert object before index.

merge(*unique=True*)

Returns the *CubeList* resulting from merging this *CubeList*.

Kwargs:

- **unique:** If True, raises *iris.exceptions.DuplicateDataError* if duplicate cubes are detected.

This combines cubes with different values of an auxiliary scalar coordinate, by constructing a new dimension.

For example:

```
>>> print(c1)
some_parameter / (unknown)          (x_vals: 3)
     Dimension coordinates:
          x_vals                           x
     Scalar coordinates:
          y_vals: 100
>>> print(c2)
some_parameter / (unknown)          (x_vals: 3)
     Dimension coordinates:
          x_vals                           x
     Scalar coordinates:
          y_vals: 200
>>> cube_list = iris.cube.CubeList([c1, c2])
>>> new_cube = cube_list.merge()[0]
>>> print(new_cube)
some_parameter / (unknown)          (y_vals: 2; x_vals: 3)
     Dimension coordinates:
          y_vals                           x          -
          x_vals                           -          x
>>> print(new_cube.coord('y_vals').points)
[100 200]
>>>
```

Contrast this with *iris.cube.CubeList.concatenate()*, which joins cubes along an existing dimension.

Note: Cubes may contain additional dimensional elements such as auxiliary coordinates, cell measures or ancillary variables. A group of similar cubes can only merge to a single result if all such elements are identical in every input cube : they

are then present, unchanged, in the merged output cube.

Note: If time coordinates in the list of cubes have differing epochs then the cubes will not be able to be merged. If this occurs, use *iris.util.unify_time_units()* to normalise the epochs of the time coordinates so that the cubes can be merged.

---

merge_cube()

Return the merged contents of the *CubeList* as a single *Cube*.

If it is not possible to merge the *CubeList* into a single *Cube*, a *MergeError* will be raised describing the reason for the failure.

For example:

```
>>> cube_1 = iris.cube.Cube([1, 2])
>>> cube_1.add_aux_coord(iris.coords.AuxCoord(0, long_name='x
↪'))
>>> cube_2 = iris.cube.Cube([3, 4])
>>> cube_2.add_aux_coord(iris.coords.AuxCoord(1, long_name='x
↪'))
>>> cube_2.add_dim_coord(
...     iris.coords.DimCoord([0, 1], long_name='z'), 0)
>>> single_cube = iris.cube.CubeList([cube_1, cube_2]).merge_
↪cube()
Traceback (most recent call last):
...
iris.exceptions.MergeError: failed to merge into a single␣
↪cube.
  Coordinates in cube.dim_coords differ: z.
  Coordinate-to-dimension mapping differs for cube.dim_coords.
```

pop(*index=- 1*, */*)

Remove and return item at index (default last).

Raises IndexError if list is empty or index is out of range.

realise_data()

Fetch 'real' data for all cubes, in a shared calculation.

This computes any lazy data, equivalent to accessing each *cube.data*. However, lazy calculations and data fetches can be shared between the computations, improving performance.

For example:

```
# Form stats.
a_std = cube_a.collapsed(['x', 'y'], iris.analysis.STD_DEV)
b_std = cube_b.collapsed(['x', 'y'], iris.analysis.STD_DEV)
ab_mean_diff = (cube_b - cube_a).collapsed(['x', 'y'],
                                           iris.analysis.MEAN)
std_err = (a_std * a_std + b_std * b_std) ** 0.5
```

(continues on next page)

(continued from previous page)

```
# Compute these stats together (avoiding multiple data␣
↪passes).
CubeList([a_std, b_std, ab_mean_diff, std_err]).realise_data()
```

Note: Cubes with non-lazy data are not affected.

remove(*value*, /)
Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse()
Reverse *IN PLACE*.

sort(*, *key=None*, *reverse=False*)
Sort the list in ascending order and return None.

The sort is in-place (i.e. the list itself is modified) and stable (i.e. the order of two equal elements is maintained).

If a key function is given, apply it once to each list item and sort them, ascending or descending, according to their function values.

The reverse flag can be set to sort in descending order.

xml(*checksum=False*, *order=True*, *byteorder=True*)
Return a string of the XML that this list of cubes represents.

27.9 iris.exceptions

Exceptions specific to the Iris package.

In this module:

- *AncillaryVariableNotFoundError*
- *CellMeasureNotFoundError*
- *ConcatenateError*
- *ConstraintMismatchError*
- *CoordinateCollapseError*
- *CoordinateMultiDimError*
- *CoordinateNotFoundError*
- *CoordinateNotRegularError*
- *DuplicateDataError*
- *IgnoreCubeException*
- *InvalidCubeError*
- *IrisError*
- *LazyAggregatorError*

- *MergeError*
- *NotYetImplementedError*
- *TranslationError*
- *UnitConversionError*

Raised when a search yields no ancillary variables.

class iris.exceptions.**AncillaryVariableNotFoundError**
Raised when a search yields no ancillary variables.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

args

Raised when a search yields no cell measures.

class iris.exceptions.**CellMeasureNotFoundError**
Raised when a search yields no cell measures.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

args

Raised when concatenate is expected to produce a single cube, but fails to do so.

class iris.exceptions.**ConcatenateError**(*differences*)
Creates a ConcatenateError with a list of textual descriptions of the differences which prevented a concatenate.

Args:

- **differences:** The list of strings which describe the differences.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

args

Raised when a constraint operation has failed to find the correct number of results.

class iris.exceptions.**ConstraintMismatchError**
Raised when a constraint operation has failed to find the correct number of results.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

args

Raised when a requested coordinate cannot be collapsed.

class iris.exceptions.**CoordinateCollapseError**
Raised when a requested coordinate cannot be collapsed.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

args

Raised when a routine doesn't support multi-dimensional coordinates.

class iris.exceptions.**CoordinateMultiDimError**(*msg*)
Raised when a routine doesn't support multi-dimensional coordinates.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

args

Raised when a search yields no coordinates.

class iris.exceptions.**CoordinateNotFoundError**
Raised when a search yields no coordinates.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

args

Raised when a coordinate is unexpectedly irregular.

class iris.exceptions.**CoordinateNotRegularError**
Raised when a coordinate is unexpectedly irregular.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

args

Raised when merging two or more cubes that have identical metadata.

class iris.exceptions.**DuplicateDataError**(*msg*)
Raised when merging two or more cubes that have identical metadata.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

args

Raised from a callback function when a cube should be ignored on load.

class iris.exceptions.**IgnoreCubeException**
Raised from a callback function when a cube should be ignored on load.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

args

Raised when a Cube validation check fails.

class iris.exceptions.**InvalidCubeError**
Raised when a Cube validation check fails.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

args

Base class for errors in the Iris package.

class iris.exceptions.**IrisError**
Base class for errors in the Iris package.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

args

Common base class for all non-exit exceptions.

class iris.exceptions.**LazyAggregatorError**

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

args

Raised when merge is expected to produce a single cube, but fails to do so.

class iris.exceptions.**MergeError**(*differences*)
Creates a MergeError with a list of textual descriptions of the differences which prevented a merge.

Args:

- **differences:** The list of strings which describe the differences.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

args

Raised by missing functionality.

Different meaning to NotImplementedError, which is for abstract methods.

class iris.exceptions.**NotYetImplementedError**
Raised by missing functionality.

Different meaning to NotImplementedError, which is for abstract methods.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

args

Raised when Iris is unable to translate format-specific codes.

class iris.exceptions.**TranslationError**
Raised when Iris is unable to translate format-specific codes.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

args

Raised when Iris is unable to convert a unit.

class iris.exceptions.**UnitConversionError**
Raised when Iris is unable to convert a unit.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

args

27.10 iris.experimental

27.10.1 iris.experimental.animate

Wrapper for animating iris cubes using iris or matplotlib plotting functions

In this module:

- *animate*

iris.experimental.animate.**animate**(*cube_iterator*, *plot_func*, *fig=None*, ***kwargs*)
Animates the given cube iterator.

Args:

- **cube_iterator (iterable of *iris.cube.Cube* objects):** Each animation frame corresponds to each *iris.cube.Cube* object. See *iris.cube.Cube.slices()*.
- **plot_func (*iris.plot* or *iris.quickplot* plotting function):** Plotting function used to animate. Must accept the signature `plot_func(cube, vmin=vmin, vmax=vmax, coords=coords)`. *contourf()*, *contour()*, *pcolor()* and *pcolormesh()* all conform to this signature.

Kwargs:

- **fig (matplotlib.figure.Figure instance):** By default, the current figure will be used or a new figure instance created if no figure is available. See matplotlib.pyplot.gcf().
- **coords (list of *Coord* objects or coordinate names):** Use the given coordinates as the axes for the plot. The order of the given coordinates indicates which axis to use for each, where the first element is the horizontal axis of the plot and the second element is the vertical axis of the plot.

- **interval (int, float or long):** Defines the time interval in milliseconds between successive frames. A default interval of 100ms is set.
- **vmin, vmax (int, float or long):** Color scaling values, see `matplotlib.colors.Normalize` for further details. Default values are determined by the min-max across the data set over the entire sequence.

See `matplotlib.animation.FuncAnimation` for details of other valid keyword arguments.

Returns `FuncAnimation` object suitable for saving and or plotting.

For example, to animate along a set of cube slices:

```
cube_iter = cubes.slices(('grid_longitude', 'grid_latitude'))
ani = animate(cube_iter, qplt.contourf)
plt.show()
```

27.10.2 iris.experimental.equalise_cubes

Experimental cube-adjusting functions to assist merge operations.

In this module:

- *equalise_attributes*

iris.experimental.equalise_cubes.**equalise_attributes**(*cubes*)
Delete cube attributes that are not identical over all cubes in a group.

> **Warning:** This function is now **disabled**.
>
> The functionality has been moved to *iris.util.equalise_attributes()*.

27.10.3 iris.experimental.regrid

Regridding functions.

In this module:

- *regrid_area_weighted_rectilinear_src_and_grid*
- *regrid_weighted_curvilinear_to_rectilinear*
- *PointInCell*
- *ProjectedUnstructuredLinear*
- *ProjectedUnstructuredNearest*

iris.experimental.regrid.**regrid_area_weighted_rectilinear_src_and_grid**(*src_cube*, *grid_cube*, *mdtol=0*)
Return a new cube with data values calculated using the area weighted mean of data values from src_grid regridded onto the horizontal grid of grid_cube.

This function requires that the horizontal grids of both cubes are rectilinear (i.e. expressed in terms of two orthogonal 1D coordinates) and that these grids are in the same

coordinate system. This function also requires that the coordinates describing the horizontal grids all have bounds.

Note: Elements in data array of the returned cube that lie either partially or entirely outside of the horizontal extent of the src_cube will be masked irrespective of the value of mdtol.

Args:

- **src_cube:** An instance of *iris.cube.Cube* that supplies the data, metadata and coordinates.
- **grid_cube:** An instance of *iris.cube.Cube* that supplies the desired horizontal grid definition.

Kwargs:

- **mdtol:** Tolerance of missing data. The value returned in each element of the returned cube's data array will be masked if the fraction of masked data in the overlapping cells of the source cube exceeds mdtol. This fraction is calculated based on the area of masked cells within each target cell. mdtol=0 means no missing data is tolerated while mdtol=1 will mean the resulting element will be masked if and only if all the overlapping cells of the source cube are masked. Defaults to 0.

Returns A new *iris.cube.Cube* instance.

---

iris.experimental.regrid.**regrid_weighted_curvilinear_to_rectilinear**(*src_cube*, *weights*, *grid_cube*)

Return a new cube with the data values calculated using the weighted mean of data values from `src_cube` and the weights from `weights` regridded onto the horizontal grid of `grid_cube`.

This function requires that the `src_cube` has a horizontal grid defined by a pair of X- and Y-axis coordinates which are mapped over the same cube dimensions, thus each point has an individually defined X and Y coordinate value. The actual dimensions of these coordinates are of no significance. The `src_cube` grid cube must have a normal horizontal grid, i.e. expressed in terms of two orthogonal 1D horizontal coordinates. Both grids must be in the same coordinate system, and the `grid_cube` must have horizontal coordinates that are both bounded and contiguous.

Note that, for any given target `grid_cube` cell, only the points from the `src_cube` that are bound by that cell will contribute to the cell result. The bounded extent of the `src_cube` will not be considered here.

A target `grid_cube` cell result will be calculated as, $\sum(src_cube.data_{ij} * weights_{ij}) / \sum weights_{ij}$, for all ij `src_cube` points that are bound by that cell.

Warning:

- All coordinates that span the `src_cube` that don't define the horizontal curvilinear grid will be ignored.

Args:

- **src_cube:** A *iris.cube.Cube* instance that defines the source variable grid to be regridded.
- **weights (array or None):** A numpy.ndarray instance that defines the weights for the source variable grid cells. Must have the same shape as the X and Y coordinates. If weights is None, all-ones will be used.
- **grid_cube:** A *iris.cube.Cube* instance that defines the target rectilinear grid.

Returns A *iris.cube.Cube* instance.

This class describes the point-in-cell regridding scheme for use typically with *iris.cube.Cube.regrid()*.

> **Warning:** This class is now **disabled**.
>
> The functionality has been moved to *iris.analysis.PointInCell*.

class iris.experimental.regrid.**PointInCell**(*weights=None*)

Point-in-cell regridding scheme suitable for regridding over one or more orthogonal coordinates.

> **Warning:** This class is now **disabled**.
>
> The functionality has been moved to *iris.analysis.PointInCell*.

This class describes the linear regridding scheme which uses the scipy.interpolate.griddata to regrid unstructured data on to a grid.

The source cube and the target cube will be projected into a common projection for the scipy calculation to be performed.

class iris.experimental.regrid.**ProjectedUnstructuredLinear**(*projection=None*)

Linear regridding scheme that uses scipy.interpolate.griddata on projected unstructured data.

Optional Args:

- **projection:** ***cartopy.crs instance*** The projection that the scipy calculation is performed in. If None is given, a PlateCarree projection is used. Defaults to None.

regridder(*src_cube*, *target_grid*)

Creates a linear regridder to perform regridding, using scipy.interpolate.griddata from unstructured source points to the target grid. The regridding calculation is performed in the given projection.

Typically you should use *iris.cube.Cube.regrid()* for regridding a cube. There are, however, some situations when constructing your own regridder is preferable. These are detailed in the *user guide*.

Does not support lazy regridding.

Args:

- **src_cube:** The *Cube* defining the unstructured source points.
- **target_grid:** The *Cube* defining the target grid.

Returns

callable(cube)

where *cube* is a cube with the same grid as *src_cube* that is to be regridded to the *target_grid*.

Return type A callable with the interface

This class describes the nearest regridding scheme which uses the scipy.interpolate.griddata to regrid unstructured data on to a grid.

The source cube and the target cube will be projected into a common projection for the scipy calculation to be performed.

Note: The *iris.analysis.UnstructuredNearest* scheme performs essentially the same job. That calculation is more rigorously correct and may be applied to larger data regions (including global). This one however, where applicable, is substantially faster.

---

class iris.experimental.regrid.**ProjectedUnstructuredNearest**(*projection=None*)

Nearest regridding scheme that uses scipy.interpolate.griddata on projected unstructured data.

Optional Args:

- **projection:** ***cartopy.crs instance*** The projection that the scipy calculation is performed in. If None is given, a PlateCarree projection is used. Defaults to None.

regridder(*src_cube*, *target_grid*)

Creates a nearest-neighbour regridder to perform regridding, using scipy.interpolate.griddata from unstructured source points to the target grid. The regridding calculation is performed in the given projection.

Typically you should use *iris.cube.Cube.regrid()* for regridding a cube. There are, however, some situations when constructing your own regridder is preferable. These are detailed in the *user guide*.

Does not support lazy regridding.

Args:

- **src_cube:** The *Cube* defining the unstructured source points.
- **target_grid:** The *Cube* defining the target grid.

Returns

callable(cube)

where *cube* is a cube with the same grid as *src_cube* that is to be regridded to the *target_grid*.

Return type A callable with the interface

27.10.4 iris.experimental.regrid_conservative

Support for conservative regridding via ESMPy.

In this module:

- *regrid_conservative_via_esmpy*

iris.experimental.regrid_conservative.**regrid_conservative_via_esmpy**(*source_cube*, *grid_cube*)

Perform a conservative regridding with ESMPy.

Regrids the data of a source cube onto a new grid defined by a destination cube.

Args:

- **source_cube (*iris.cube.Cube*):** Source data. Must have two identifiable horizontal dimension coordinates.
- **grid_cube (*iris.cube.Cube*):** Define the target horizontal grid: Only the horizontal dimension coordinates are actually used.

Returns A new cube derived from source_cube, regridded onto the specified horizontal grid.

Any additional coordinates which map onto the horizontal dimensions are removed, while all other metadata is retained. If there are coordinate factories with 2d horizontal reference surfaces, the reference surfaces are also regridded, using ordinary bilinear interpolation.

Note: Both source and destination cubes must have two dimension coordinates identified with axes 'X' and 'Y' which share a coord_system with a Cartopy CRS. The grids are defined by *iris.coords.Coord.contiguous_bounds()* of these.

Note: Initialises the ESMF Manager, if it was not already called. This implements default Manager operations (e.g. logging).

To alter this, make a prior call to ESMF.Manager().

27.10.5 iris.experimental.representation

Definitions of how Iris objects should be represented.

In this module:

- *CubeListRepresentation*
- *CubeRepresentation*

None

class iris.experimental.representation.**CubeListRepresentation**(*cubelist*)

make_content()

repr_html()

Produce representations of a *Cube*.

This includes:

- `_html_repr_`: a representation of the cube as an html object, available in Jupyter notebooks. Specifically, this is presented as an html table.

  class iris.experimental.representation.**CubeRepresentation**(*cube*)
  Produce representations of a *Cube*.

  This includes:
  - `_html_repr_`: a representation of the cube as an html object, available in Jupyter notebooks. Specifically, this is presented as an html table.

  repr_html()
  The *repr* interface for Jupyter.

27.10.6 iris.experimental.stratify

Routines for putting data on new strata (aka. isosurfaces), often in the Z direction.

In this module:

- *relevel*

iris.experimental.stratify.**relevel**(*cube*, *src_levels*, *tgt_levels*, *axis=None*, *interpolator=None*)
Interpolate the cube onto the specified target levels, given the source levels of the cube.

For example, suppose we have two datasets *P(i,j,k)* and *H(i,j,k)* and we want *P(i,j,H)*. We call *relevel()* with *cube=P*, *src_levels=H* and *tgt_levels* being an array of the values of *H* we would like.

This routine is especially useful for computing isosurfaces of phenomenon that are generally monotonic in the direction of interpolation, such as height/pressure or salinity/depth.

Args:

cube [*Cube*] The phenomenon data to be re-levelled.

src_levels [*Cube*, Coord or string] Describes the source levels of the *cube* that will be interpolated over. The *src_levels* must be in the same system as the *tgt_levels*. The dimensions of *src_levels* must be broadcastable to the dimensions of the *cube*. Note that, the coordinate name containing the source levels in the *cube* may be provided.

tgt_levels [array-like] Describes the target levels of the *cube* to be interpolated to. The *tgt_levels* must be in the same system as the *src_levels*. The dimensions of the *tgt_levels* must be broadcastable to the dimensions of the *cube*, except in the nominated axis of interpolation.

axis [int, *Coord* or string] The axis of interpolation. Defaults to the first dimension of the *cube*, which is typically the z-dimension. Note that, the coordinate name specifying the z-dimension of the *cube* may be provided.

interpolator [callable or None] The interpolator to use when computing the interpolation. The function will be passed the following positional arguments:

```
(tgt-data, src-data, cube-data, axis-of-interpolation)
```

If the interpolator is None, `stratify.interpolate()` will be used with linear interpolation and NaN extrapolation.

An example of constructing an alternative interpolation scheme:

```
from functools import partial
interpolator = partial(stratify.interpolate,
                       interpolation=stratify.INTERPOLATE_
↪NEAREST,
                       extrapolation=stratify.EXTRAPOLATE_
↪LINEAR)
```

27.10.7 iris.experimental.ugrid

Ugrid functions.

In this module:

- *ugrid*

iris.experimental.ugrid.**ugrid**(*location*, *name*)
Create a cube from an unstructured grid.

Args:

- **location:** A string whose value represents the path to a file or URL to an OpenDAP resource conforming to the Unstructured Grid Metadata Conventions for Scientific Datasets https://github.com/ugrid-conventions/ugrid-conventions
- **name:** A string whose value represents a cube loading constraint of first the standard name if found, then the long name if found, then the variable name if found, before falling back to the value of the default which itself defaults to "unknown"

Returns An instance of *iris.cube.Cube* decorated with an instance of `pyugrid.ugrid.Ugrid` bound to an attribute of the cube called "mesh"

Experimental code can be introduced to Iris through this package.

Changes to experimental code may be more extensive than in the rest of the codebase. The code is expected to graduate, eventually, to "full status".

In this module:

27.11 iris.fileformats

27.11.1 iris.fileformats.abf

Provides ABF (and ABL) file format capabilities.

ABF and ABL files are satellite file formats defined by Boston University. Including this module adds ABF and ABL loading to the session's capabilities.

The documentation for this file format can be found here.

In this module:

- *load_cubes*
- *ABFField*

iris.fileformats.abf.**load_cubes**(*filespecs*, *callback=None*)
Loads cubes from a list of ABF filenames.

Args:

- filenames - list of ABF filenames to load

Kwargs:

- callback - a function that can be passed to *iris.io.run_callback()*

Note: The resultant cubes may not be in the same order as in the file.

A data field from an ABF (or ABL) file.

Capable of creating a *Cube*.

class iris.fileformats.abf.**ABFField**(*filename*)
Create an ABFField object from the given filename.
Parameters **filename.** (** filename - An ABF*) –
Example:

```
field = ABFField("AVHRRBUVI01.1985feba.abl")
```

to_cube()
Return a new *Cube* from this ABFField.

27.11.2 iris.fileformats.cf

Provides the capability to load netCDF files and interpret them according to the 'NetCDF Climate and Forecast (CF) Metadata Conventions'.

References:

[CF] NetCDF Climate and Forecast (CF) Metadata conventions. [NUG] NetCDF User's Guide, https://www.unidata.ucar.edu/software/netcdf/documentation/NUG/

In this module:

- *CFAncillaryDataVariable*
- *CFAuxiliaryCoordinateVariable*
- *CFBoundaryVariable*
- *CFClimatologyVariable*
- *CFCoordinateVariable*
- *CFDataVariable*
- *CFGridMappingVariable*
- *CFGroup*
- *CFLabelVariable*

- *CFMeasureVariable*
- *CFReader*
- *CFVariable*

A CF-netCDF ancillary data variable is a variable that provides metadata about the individual values of another data variable.

Identified by the CF-netCDF variable attribute 'ancillary_variables'.

Ref: [CF] Section 3.4. Ancillary Data.

class iris.fileformats.cf.**CFAncillaryDataVariable**(*name*, *data*)

A CF-netCDF ancillary data variable is a variable that provides metadata about the individual values of another data variable.

Identified by the CF-netCDF variable attribute 'ancillary_variables'.

Ref: [CF] Section 3.4. Ancillary Data.

add_formula_term(*root*, *term*)

Register the participation of this CF-netCDF variable in a CF-netCDF formula term.

Args:

- **root (string):** The name of CF-netCDF variable that defines the CF-netCDF formula_terms attribute.
- **term (string):** The associated term name of this variable in the formula_terms definition.

Returns None.

cf_attrs()

Return a list of all attribute name and value pairs of the CF-netCDF variable.

cf_attrs_ignored()

Return a list of all ignored attribute name and value pairs of the CF-netCDF variable.

cf_attrs_reset()

Reset the history of accessed attribute names of the CF-netCDF variable.

cf_attrs_unused()

Return a list of all non-accessed attribute name and value pairs of the CF-netCDF variable.

cf_attrs_used()

Return a list of all accessed attribute name and value pairs of the CF-netCDF variable.

has_formula_terms()

Determine whether this CF-netCDF variable participates in a CF-netcdf formula term.

Returns Boolean.

classmethod identify(*variables*, *ignore=None*, *target=None*, *warn=True*)

spans(*cf_variable*)

Determine whether the dimensionality of this variable is a subset of the specified target variable.

Note that, by default scalar variables always span the dimensionality of the target variable.

Args:
- **cf_variable:** Compare dimensionality with the *CFVariable*.

Returns Boolean.

cf_data
NetCDF4 Variable data instance.

cf_group
Collection of CF-netCDF variables associated with this variable.

cf_identity = 'ancillary_variables'
Name of the netCDF variable attribute that identifies this CF-netCDF variable.

cf_name
NetCDF variable name.

cf_terms_by_root
CF-netCDF formula terms that his variable participates in.

A CF-netCDF auxiliary coordinate variable is any netCDF variable that contains coordinate data, but is not a CF-netCDF coordinate variable by definition.

There is no relationship between the name of a CF-netCDF auxiliary coordinate variable and the name(s) of its dimension(s).

Identified by the CF-netCDF variable attribute 'coordinates'. Also see *iris.fileformats.cf.CFLabelVariable*.

Ref: [CF] Chapter 5. Coordinate Systems. [CF] Section 6.2. Alternative Coordinates.

class iris.fileformats.cf.**CFAuxiliaryCoordinateVariable**(*name*, *data*)

A CF-netCDF auxiliary coordinate variable is any netCDF variable that contains coordinate data, but is not a CF-netCDF coordinate variable by definition.

There is no relationship between the name of a CF-netCDF auxiliary coordinate variable and the name(s) of its dimension(s).

Identified by the CF-netCDF variable attribute 'coordinates'. Also see *iris.fileformats.cf.CFLabelVariable*.

Ref: [CF] Chapter 5. Coordinate Systems. [CF] Section 6.2. Alternative Coordinates.

add_formula_term(*root*, *term*)
Register the participation of this CF-netCDF variable in a CF-netCDF formula term.

Args:
- **root (string):** The name of CF-netCDF variable that defines the CF-netCDF formula_terms attribute.
- **term (string):** The associated term name of this variable in the formula_terms definition.

Returns None.

cf_attrs()
Return a list of all attribute name and value pairs of the CF-netCDF variable.

cf_attrs_ignored()
Return a list of all ignored attribute name and value pairs of the CF-netCDF variable.

cf_attrs_reset()
Reset the history of accessed attribute names of the CF-netCDF variable.

cf_attrs_unused()
Return a list of all non-accessed attribute name and value pairs of the CF-netCDF variable.

cf_attrs_used()
Return a list of all accessed attribute name and value pairs of the CF-netCDF variable.

has_formula_terms()
Determine whether this CF-netCDF variable participates in a CF-netcdf formula term.

Returns Boolean.

classmethod identify(*variables*, *ignore=None*, *target=None*, *warn=True*)

spans(*cf_variable*)
Determine whether the dimensionality of this variable is a subset of the specified target variable.

Note that, by default scalar variables always span the dimensionality of the target variable.

Args:

- **cf_variable:** Compare dimensionality with the *CFVariable*.

Returns Boolean.

cf_data
NetCDF4 Variable data instance.

cf_group
Collection of CF-netCDF variables associated with this variable.

cf_identity = 'coordinates'
Name of the netCDF variable attribute that identifies this CF-netCDF variable.

cf_name
NetCDF variable name.

cf_terms_by_root
CF-netCDF formula terms that his variable participates in.

A CF-netCDF boundary variable is associated with a CF-netCDF variable that contains coordinate data. When a data value provides information about conditions in a cell occupying a region of space/time or some other dimension, the boundary variable provides a description of cell extent.

A CF-netCDF boundary variable will have one more dimension than its associated CF-netCDF coordinate variable or CF-netCDF auxiliary coordinate variable.

Identified by the CF-netCDF variable attribute 'bounds'.

Ref: [CF] Section 7.1. Cell Boundaries.

class iris.fileformats.cf.**CFBoundaryVariable**(*name*, *data*)

A CF-netCDF boundary variable is associated with a CF-netCDF variable that contains coordinate data. When a data value provides information about conditions in a cell occupying a region of space/time or some other dimension, the boundary variable provides a description of cell extent.

A CF-netCDF boundary variable will have one more dimension than its associated CF-netCDF coordinate variable or CF-netCDF auxiliary coordinate variable.

Identified by the CF-netCDF variable attribute 'bounds'.

Ref: [CF] Section 7.1. Cell Boundaries.

add_formula_term(*root*, *term*)

Register the participation of this CF-netCDF variable in a CF-netCDF formula term.

Args:

- **root (string):** The name of CF-netCDF variable that defines the CF-netCDF formula_terms attribute.
- **term (string):** The associated term name of this variable in the formula_terms definition.

Returns None.

cf_attrs()

Return a list of all attribute name and value pairs of the CF-netCDF variable.

cf_attrs_ignored()

Return a list of all ignored attribute name and value pairs of the CF-netCDF variable.

cf_attrs_reset()

Reset the history of accessed attribute names of the CF-netCDF variable.

cf_attrs_unused()

Return a list of all non-accessed attribute name and value pairs of the CF-netCDF variable.

cf_attrs_used()

Return a list of all accessed attribute name and value pairs of the CF-netCDF variable.

has_formula_terms()

Determine whether this CF-netCDF variable participates in a CF-netcdf formula term.

Returns Boolean.

classmethod identify(*variables*, *ignore=None*, *target=None*, *warn=True*)

spans(*cf_variable*)
Determine whether the dimensionality of this variable is a subset of the specified target variable.

Note that, by default scalar variables always span the dimensionality of the target variable.

Args:
- **cf_variable:** Compare dimensionality with the *CFVariable*.

Returns Boolean.

cf_data
NetCDF4 Variable data instance.

cf_group
Collection of CF-netCDF variables associated with this variable.

cf_identity = 'bounds'
Name of the netCDF variable attribute that identifies this CF-netCDF variable.

cf_name
NetCDF variable name.

cf_terms_by_root
CF-netCDF formula terms that his variable participates in.

A CF-netCDF climatology variable is associated with a CF-netCDF variable that contains coordinate data. When a data value provides information about conditions in a cell occupying a region of space/time or some other dimension, the climatology variable provides a climatological description of cell extent.

A CF-netCDF climatology variable will have one more dimension than its associated CF-netCDF coordinate variable.

Identified by the CF-netCDF variable attribute 'climatology'.

Ref: [CF] Section 7.4. Climatological Statistics

class iris.fileformats.cf.**CFClimatologyVariable**(*name*, *data*)
A CF-netCDF climatology variable is associated with a CF-netCDF variable that contains coordinate data. When a data value provides information about conditions in a cell occupying a region of space/time or some other dimension, the climatology variable provides a climatological description of cell extent.

A CF-netCDF climatology variable will have one more dimension than its associated CF-netCDF coordinate variable.

Identified by the CF-netCDF variable attribute 'climatology'.

Ref: [CF] Section 7.4. Climatological Statistics

add_formula_term(*root*, *term*)
Register the participation of this CF-netCDF variable in a CF-netCDF formula term.

Args:
- **root (string):** The name of CF-netCDF variable that defines the CF-netCDF formula_terms attribute.

- **term (string):** The associated term name of this variable in the formula_terms definition.

Returns None.

cf_attrs()
Return a list of all attribute name and value pairs of the CF-netCDF variable.

cf_attrs_ignored()
Return a list of all ignored attribute name and value pairs of the CF-netCDF variable.

cf_attrs_reset()
Reset the history of accessed attribute names of the CF-netCDF variable.

cf_attrs_unused()
Return a list of all non-accessed attribute name and value pairs of the CF-netCDF variable.

cf_attrs_used()
Return a list of all accessed attribute name and value pairs of the CF-netCDF variable.

has_formula_terms()
Determine whether this CF-netCDF variable participates in a CF-netcdf formula term.
Returns Boolean.

classmethod identify(*variables*, *ignore=None*, *target=None*, *warn=True*)

spans(*cf_variable*)
Determine whether the dimensionality of this variable is a subset of the specified target variable.

Note that, by default scalar variables always span the dimensionality of the target variable.

Args:
- **cf_variable:** Compare dimensionality with the *CFVariable*.

Returns Boolean.

cf_data
NetCDF4 Variable data instance.

cf_group
Collection of CF-netCDF variables associated with this variable.

cf_identity = 'climatology'
Name of the netCDF variable attribute that identifies this CF-netCDF variable.

cf_name
NetCDF variable name.

cf_terms_by_root
CF-netCDF formula terms that his variable participates in.

A CF-netCDF coordinate variable is a one-dimensional variable with the same name as its dimension, and it is defined as a numeric data type with values that are ordered monotonically.

Missing values are not allowed in CF-netCDF coordinate variables. Also see [NUG] Section 2.3.1.

Identified by the above criterion, there is no associated CF-netCDF variable attribute.

Ref: [CF] 1.2. Terminology.

class iris.fileformats.cf.**CFCoordinateVariable**(*name*, *data*)

A CF-netCDF coordinate variable is a one-dimensional variable with the same name as its dimension, and it is defined as a numeric data type with values that are ordered monotonically. Missing values are not allowed in CF-netCDF coordinate variables. Also see [NUG] Section 2.3.1.

Identified by the above criterion, there is no associated CF-netCDF variable attribute.

Ref: [CF] 1.2. Terminology.

add_formula_term(*root*, *term*)

Register the participation of this CF-netCDF variable in a CF-netCDF formula term.

Args:

- **root (string):** The name of CF-netCDF variable that defines the CF-netCDF formula_terms attribute.
- **term (string):** The associated term name of this variable in the formula_terms definition.

Returns None.

cf_attrs()

Return a list of all attribute name and value pairs of the CF-netCDF variable.

cf_attrs_ignored()

Return a list of all ignored attribute name and value pairs of the CF-netCDF variable.

cf_attrs_reset()

Reset the history of accessed attribute names of the CF-netCDF variable.

cf_attrs_unused()

Return a list of all non-accessed attribute name and value pairs of the CF-netCDF variable.

cf_attrs_used()

Return a list of all accessed attribute name and value pairs of the CF-netCDF variable.

has_formula_terms()

Determine whether this CF-netCDF variable participates in a CF-netcdf formula term.

Returns Boolean.

classmethod identify(*variables*, *ignore=None*, *target=None*, *warn=True*, *monotonic=False*)

spans(*cf_variable*)

Determine whether the dimensionality of this variable is a subset of the specified target variable.

Note that, by default scalar variables always span the dimensionality of the target variable.

Args:
- **cf_variable:** Compare dimensionality with the *CFVariable*.

Returns Boolean.

cf_data
NetCDF4 Variable data instance.

cf_group
Collection of CF-netCDF variables associated with this variable.

cf_identity = None
Name of the netCDF variable attribute that identifies this CF-netCDF variable.

cf_name
NetCDF variable name.

cf_terms_by_root
CF-netCDF formula terms that his variable participates in.

A CF-netCDF variable containing data pay-load that maps to an Iris *iris.cube.Cube*.

class iris.fileformats.cf.**CFDataVariable**(*name*, *data*)
A CF-netCDF variable containing data pay-load that maps to an Iris *iris.cube.Cube*.

add_formula_term(*root*, *term*)
Register the participation of this CF-netCDF variable in a CF-netCDF formula term.

Args:
- **root (string):** The name of CF-netCDF variable that defines the CF-netCDF formula_terms attribute.
- **term (string):** The associated term name of this variable in the formula_terms definition.

Returns None.

cf_attrs()
Return a list of all attribute name and value pairs of the CF-netCDF variable.

cf_attrs_ignored()
Return a list of all ignored attribute name and value pairs of the CF-netCDF variable.

cf_attrs_reset()
Reset the history of accessed attribute names of the CF-netCDF variable.

cf_attrs_unused()
Return a list of all non-accessed attribute name and value pairs of the CF-netCDF variable.

cf_attrs_used()
Return a list of all accessed attribute name and value pairs of the CF-netCDF variable.

has_formula_terms()
Determine whether this CF-netCDF variable participates in a CF-netcdf formula term.
Returns Boolean.

classmethod identify(*variables*, *ignore=None*, *target=None*, *warn=True*)

spans(*cf_variable*)
Determine whether the dimensionality of this variable is a subset of the specified target variable.

Note that, by default scalar variables always span the dimensionality of the target variable.

Args:
- **cf_variable:** Compare dimensionality with the *CFVariable*.

Returns Boolean.

cf_data
NetCDF4 Variable data instance.

cf_group
Collection of CF-netCDF variables associated with this variable.

cf_identity = None
Name of the netCDF variable attribute that identifies this CF-netCDF variable.

cf_name
NetCDF variable name.

cf_terms_by_root
CF-netCDF formula terms that his variable participates in.

A CF-netCDF grid mapping variable contains a list of specific attributes that define a particular grid mapping. A CF-netCDF grid mapping variable must contain the attribute 'grid_mapping_name'.

Based on the value of the 'grid_mapping_name' attribute, there are associated standard names of CF-netCDF coordinate variables that contain the mapping's independent variables.

Identified by the CF-netCDF variable attribute 'grid_mapping'.

Ref: [CF] Section 5.6. Horizontal Coordinate Reference Systems, Grid Mappings, and Projections. [CF] Appendix F. Grid Mappings.

class iris.fileformats.cf.**CFGridMappingVariable**(*name*, *data*)
A CF-netCDF grid mapping variable contains a list of specific attributes that define a particular grid mapping. A CF-netCDF grid mapping variable must contain the attribute 'grid_mapping_name'.

Based on the value of the 'grid_mapping_name' attribute, there are associated standard names of CF-netCDF coordinate variables that contain the mapping's independent variables.

Identified by the CF-netCDF variable attribute 'grid_mapping'.
Ref: [CF] Section 5.6. Horizontal Coordinate Reference Systems, Grid Mappings, and Projections. [CF] Appendix F. Grid Mappings.

add_formula_term(*root*, *term*)
Register the participation of this CF-netCDF variable in a CF-netCDF formula term.

Args:
- **root (string):** The name of CF-netCDF variable that defines the CF-netCDF formula_terms attribute.
- **term (string):** The associated term name of this variable in the formula_terms definition.

Returns None.

cf_attrs()
Return a list of all attribute name and value pairs of the CF-netCDF variable.

cf_attrs_ignored()
Return a list of all ignored attribute name and value pairs of the CF-netCDF variable.

cf_attrs_reset()
Reset the history of accessed attribute names of the CF-netCDF variable.

cf_attrs_unused()
Return a list of all non-accessed attribute name and value pairs of the CF-netCDF variable.

cf_attrs_used()
Return a list of all accessed attribute name and value pairs of the CF-netCDF variable.

has_formula_terms()
Determine whether this CF-netCDF variable participates in a CF-netcdf formula term.
Returns Boolean.

classmethod identify(*variables*, *ignore=None*, *target=None*, *warn=True*)

spans(*cf_variable*)
Determine whether the dimensionality of this variable is a subset of the specified target variable.

Note that, by default scalar variables always span the dimensionality of the target variable.

Args:
- **cf_variable:** Compare dimensionality with the *CFVariable*.

Returns Boolean.

cf_data
NetCDF4 Variable data instance.

cf_group
Collection of CF-netCDF variables associated with this variable.

cf_identity = 'grid_mapping'
Name of the netCDF variable attribute that identifies this CF-netCDF variable.

cf_name
NetCDF variable name.

cf_terms_by_root
CF-netCDF formula terms that his variable participates in.

Represents a collection of 'NetCDF Climate and Forecast (CF) Metadata Conventions' variables and netCDF global attributes.

class iris.fileformats.cf.**CFGroup**
Represents a collection of 'NetCDF Climate and Forecast (CF) Metadata Conventions' variables and netCDF global attributes.

clear() → None. Remove all items from D.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys()
Return the names of all the CF-netCDF variables in the group.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

property ancillary_variables
Collection of CF-netCDF ancillary variables.

property auxiliary_coordinates
Collection of CF-netCDF auxiliary coordinate variables.

property bounds
Collection of CF-netCDF boundary variables.

property cell_measures
Collection of CF-netCDF measure variables.

property climatology
Collection of CF-netCDF climatology variables.

property coordinates
Collection of CF-netCDF coordinate variables.

property data_variables
Collection of CF-netCDF data pay-load variables.

property formula_terms
Collection of CF-netCDF variables that participate in a CF-netCDF formula term.

global_attributes
Collection of netCDF global attributes

property grid_mappings
Collection of CF-netCDF grid mapping variables.

property labels
Collection of CF-netCDF label variables.

promoted
Collection of CF-netCDF variables promoted to a CFDataVariable.

A CF-netCDF CF label variable is any netCDF variable that contain string textual information, or labels.

Identified by the CF-netCDF variable attribute 'coordinates'. Also see *iris.fileformats.cf.CFAuxiliaryCoordinateVariable*.

Ref: [CF] Section 6.1. Labels.

class iris.fileformats.cf.**CFLabelVariable**(*name*, *data*)
A CF-netCDF CF label variable is any netCDF variable that contain string textual information, or labels.

Identified by the CF-netCDF variable attribute 'coordinates'. Also see *iris.fileformats.cf.CFAuxiliaryCoordinateVariable*.

Ref: [CF] Section 6.1. Labels.

add_formula_term(*root*, *term*)
Register the participation of this CF-netCDF variable in a CF-netCDF formula term.

Args:
- **root (string):** The name of CF-netCDF variable that defines the CF-netCDF formula_terms attribute.
- **term (string):** The associated term name of this variable in the formula_terms definition.

Returns None.

cf_attrs()
Return a list of all attribute name and value pairs of the CF-netCDF variable.

cf_attrs_ignored()
Return a list of all ignored attribute name and value pairs of the CF-netCDF variable.

cf_attrs_reset()
Reset the history of accessed attribute names of the CF-netCDF variable.

cf_attrs_unused()
Return a list of all non-accessed attribute name and value pairs of the CF-netCDF variable.

cf_attrs_used()
Return a list of all accessed attribute name and value pairs of the CF-netCDF variable.

cf_label_data(*cf_data_var*)
Return the associated CF-netCDF label variable strings.

Args:
- **cf_data_var (*iris.fileformats.cf.CFDataVariable*):** The CF-netCDF data variable which the CF-netCDF label variable describes.

Returns String labels.

cf_label_dimensions(*cf_data_var*)
Return the name of the associated CF-netCDF label variable data dimensions.

Args:
- **cf_data_var (*iris.fileformats.cf.CFDataVariable*):** The CF-netCDF data variable which the CF-netCDF label variable describes.

Returns Tuple of label data dimension names.

has_formula_terms()
Determine whether this CF-netCDF variable participates in a CF-netcdf formula term.
Returns Boolean.

classmethod identify(*variables*, *ignore=None*, *target=None*, *warn=True*)

spans(*cf_variable*)
Determine whether the dimensionality of this variable is a subset of the specified target variable.

Note that, by default scalar variables always span the dimensionality of the target variable.

Args:
- **cf_variable:** Compare dimensionality with the *CFVariable*.

Returns Boolean.

cf_data
NetCDF4 Variable data instance.

cf_group
Collection of CF-netCDF variables associated with this variable.

cf_identity = 'coordinates'
Name of the netCDF variable attribute that identifies this CF-netCDF variable.

cf_name
NetCDF variable name.

cf_terms_by_root
CF-netCDF formula terms that his variable participates in.

A CF-netCDF measure variable is a variable that contains cell areas or volumes.

Identified by the CF-netCDF variable attribute 'cell_measures'.

Ref: [CF] Section 7.2. Cell Measures.

class iris.fileformats.cf.**CFMeasureVariable**(*name*, *data*, *measure*)

A CF-netCDF measure variable is a variable that contains cell areas or volumes.

Identified by the CF-netCDF variable attribute 'cell_measures'.

Ref: [CF] Section 7.2. Cell Measures.

add_formula_term(*root*, *term*)

Register the participation of this CF-netCDF variable in a CF-netCDF formula term.

Args:

- **root (string):** The name of CF-netCDF variable that defines the CF-netCDF formula_terms attribute.
- **term (string):** The associated term name of this variable in the formula_terms definition.

Returns None.

cf_attrs()

Return a list of all attribute name and value pairs of the CF-netCDF variable.

cf_attrs_ignored()

Return a list of all ignored attribute name and value pairs of the CF-netCDF variable.

cf_attrs_reset()

Reset the history of accessed attribute names of the CF-netCDF variable.

cf_attrs_unused()

Return a list of all non-accessed attribute name and value pairs of the CF-netCDF variable.

cf_attrs_used()

Return a list of all accessed attribute name and value pairs of the CF-netCDF variable.

has_formula_terms()

Determine whether this CF-netCDF variable participates in a CF-netcdf formula term.

Returns Boolean.

classmethod identify(*variables*, *ignore=None*, *target=None*, *warn=True*)

spans(*cf_variable*)

Determine whether the dimensionality of this variable is a subset of the specified target variable.

Note that, by default scalar variables always span the dimensionality of the target variable.

Args:

- **cf_variable:** Compare dimensionality with the *CFVariable*.

Returns Boolean.

cf_data

NetCDF4 Variable data instance.

cf_group
Collection of CF-netCDF variables associated with this variable.

cf_identity = 'cell_measures'
Name of the netCDF variable attribute that identifies this CF-netCDF variable.

cf_measure
Associated cell measure of the cell variable

cf_name
NetCDF variable name.

cf_terms_by_root
CF-netCDF formula terms that his variable participates in.

This class allows the contents of a netCDF file to be interpreted according to the 'NetCDF Climate and Forecast (CF) Metadata Conventions'.

class iris.fileformats.cf.**CFReader**(*filename*, *warn=False*, *monotonic=False*)
This class allows the contents of a netCDF file to be interpreted according to the 'NetCDF Climate and Forecast (CF) Metadata Conventions'.

cf_group
Collection of CF-netCDF variables associated with this netCDF file

---

Abstract base class wrapper for a CF-netCDF variable.

class iris.fileformats.cf.**CFVariable**(*name*, *data*)
Abstract base class wrapper for a CF-netCDF variable.

add_formula_term(*root*, *term*)
Register the participation of this CF-netCDF variable in a CF-netCDF formula term.

Args:
- **root (string):** The name of CF-netCDF variable that defines the CF-netCDF formula_terms attribute.
- **term (string):** The associated term name of this variable in the formula_terms definition.

Returns None.

cf_attrs()
Return a list of all attribute name and value pairs of the CF-netCDF variable.

cf_attrs_ignored()
Return a list of all ignored attribute name and value pairs of the CF-netCDF variable.

cf_attrs_reset()
Reset the history of accessed attribute names of the CF-netCDF variable.

cf_attrs_unused()
Return a list of all non-accessed attribute name and value pairs of the CF-netCDF variable.

cf_attrs_used()
Return a list of all accessed attribute name and value pairs of the CF-netCDF variable.

has_formula_terms()
Determine whether this CF-netCDF variable participates in a CF-netcdf formula term.

Returns Boolean.

abstract identify(*variables*, *ignore=None*, *target=None*, *warn=True*)
Identify all variables that match the criterion for this CF-netCDF variable class.

Args:
- **variables:** Dictionary of netCDF4.Variable instance by variable name.

Kwargs:
- **ignore:** List of variable names to ignore.
- **target:** Name of a single variable to check.
- **warn:** Issue a warning if a missing variable is referenced.

Returns Dictionary of CFVariable instance by variable name.

spans(*cf_variable*)
Determine whether the dimensionality of this variable is a subset of the specified target variable.

Note that, by default scalar variables always span the dimensionality of the target variable.

Args:
- **cf_variable:** Compare dimensionality with the *CFVariable*.

Returns Boolean.

cf_data
NetCDF4 Variable data instance.

cf_group
Collection of CF-netCDF variables associated with this variable.

cf_identity = None
Name of the netCDF variable attribute that identifies this CF-netCDF variable.

cf_name
NetCDF variable name.

cf_terms_by_root
CF-netCDF formula terms that his variable participates in.

27.11.3 iris.fileformats.dot

Provides Creation and saving of DOT graphs for a *iris.cube.Cube*.

In this module:

- *cube_text*
- *save*
- *save_png*

iris.fileformats.dot.**cube_text**(*cube*)
Return a DOT text representation a *iris.cube.Cube*.

Parameters **text.** (** cube - The cube for which to create DOT*)
–

---

iris.fileformats.dot.**save**(*cube*, *target*)
Save a dot representation of the cube.

:param * cube - A *iris.cube.Cube*.: :param * target - A filename or open file handle.:

See also *iris.io.save()*.

---

iris.fileformats.dot.**save_png**(*source*, *target*, *launch=False*)
Produces a "dot" instance diagram by calling dot and optionally launching the resulting image.

:param * source - A *iris.cube.Cube*: :param or dot filename.: :param * target - A filename or open file handle.: If passing a file handle, take care to open it for binary output.

Kwargs:

- launch - Display the image. Default is False.

See also *iris.io.save()*.

27.11.4 iris.fileformats.name

Provides NAME file format loading capabilities.

In this module:

- *load_cubes*

iris.fileformats.name.**load_cubes**(*filenames*, *callback*)
Return a generator of cubes given one or more filenames and an optional callback.

Args:

- **filenames (string/list):** One or more NAME filenames to load.

Kwargs:

- **callback (callable function):** A function which can be passed on to *iris.io.run_callback()*.

Returns A generator of `iris.cubes.Cube` instances.

27.11.5 iris.fileformats.name_loaders

NAME file format loading functions.

In this module:

- *load_NAMEIII_field*
- *load_NAMEIII_timeseries*
- *load_NAMEIII_trajectory*
- *load_NAMEIII_version2*
- *load_NAMEII_field*
- *load_NAMEII_timeseries*
- *read_header*
- *NAMECoord*

iris.fileformats.name_loaders.**load_NAMEIII_field**(*filename*)
Load a NAME III grid output file returning a generator of *iris.cube.Cube* instances.

Args:

- **filename (string):** Name of file to load.

Returns A generator *iris.cube.Cube* instances.

---

iris.fileformats.name_loaders.**load_NAMEIII_timeseries**(*filename*)
Load a NAME III time series file returning a generator of *iris.cube.Cube* instances.

Args:

- **filename (string):** Name of file to load.

Returns A generator *iris.cube.Cube* instances.

---

iris.fileformats.name_loaders.**load_NAMEIII_trajectory**(*filename*)
Load a NAME III trajectory file returning a generator of *iris.cube.Cube* instances.

Args:

- **filename (string):** Name of file to load.

Returns A generator *iris.cube.Cube* instances.

---

iris.fileformats.name_loaders.**load_NAMEIII_version2**(*filename*)
Load a NAME III version 2 file returning a generator of *iris.cube.Cube* instances.

Args:

- **filename (string):** Name of file to load.

Returns A generator *iris.cube.Cube* instances.

---

iris.fileformats.name_loaders.**load_NAMEII_field**(*filename*)

Load a NAME II grid output file returning a generator of *iris.cube.Cube* instances.

Args:

- **filename (string):** Name of file to load.

Returns A generator *iris.cube.Cube* instances.

---

iris.fileformats.name_loaders.**load_NAMEII_timeseries**(*filename*)

Load a NAME II Time Series file returning a generator of *iris.cube.Cube* instances.

Args:

- **filename (string):** Name of file to load.

Returns A generator *iris.cube.Cube* instances.

---

iris.fileformats.name_loaders.**read_header**(*file_handle*)

Return a dictionary containing the header information extracted from the the provided NAME file object.

Args:

- **file_handle (file-like object):** A file-like object from which to read the header information.

Returns A dictionary containing the extracted header information.

NAMECoord(name, dimension, values)

class iris.fileformats.name_loaders.**NAMECoord**(*_cls*, *name*, *dimension*, *values*)

Create new instance of NAMECoord(name, dimension, values)

count(*value*, */*)

Return number of occurrences of value.

index(*value*, *start=0*, *stop=9223372036854775807*, */*)

Return first index of value.

Raises ValueError if the value is not present.

dimension

Alias for field number 1

name

Alias for field number 0

values

Alias for field number 2

27.11.6 iris.fileformats.netcdf

Module to support the loading of a NetCDF file into an Iris cube.

See also: netCDF4 python

Also refer to document 'NetCDF Climate and Forecast (CF) Metadata Conventions'.

In this module:

- *load_cubes*
- *parse_cell_methods*
- *save*
- *CFNameCoordMap*
- *NetCDFDataProxy*
- *OrderedAddableList*
- *Saver*
- *UnknownCellMethodWarning*

iris.fileformats.netcdf.**load_cubes**(*filenames*, *callback=None*, *constraints=None*)
Loads cubes from a list of NetCDF filenames/URLs.

Args:

- **filenames (string/list):** One or more NetCDF filenames/DAP URLs to load from.

Kwargs:

- **callback (callable function):** Function which can be passed on to *iris.io.run_callback()*.

Returns Generator of loaded NetCDF `iris.cubes.Cube`.

---

iris.fileformats.netcdf.**parse_cell_methods**(*nc_cell_methods*)
Parse a CF cell_methods attribute string into a tuple of zero or more CellMethod instances.

Args:

- **nc_cell_methods (str):** The value of the cell methods attribute to be parsed.

Returns:

- **cell_methods** An iterable of *iris.coords.CellMethod*.

Multiple coordinates, intervals and comments are supported. If a method has a non-standard name a warning will be issued, but the results are not affected.

---

iris.fileformats.netcdf.**save**(*cube*, *filename*, *netcdf_format='NETCDF4'*, *local_keys=None*, *unlimited_dimensions=None*, *zlib=False*, *complevel=4*, *shuffle=True*, *fletcher32=False*, *contiguous=False*, *chunksizes=None*, *endian='native'*, *least_significant_digit=None*, *packing=None*, *fill_value=None*)
Save cube(s) to a netCDF file, given the cube and the filename.

- Iris will write CF 1.7 compliant NetCDF files.
- The attributes dictionaries on each cube in the saved cube list will be compared and common attributes saved as NetCDF global attributes where appropriate.
- Keyword arguments specifying how to save the data are applied to each cube. To use different settings for different cubes, use the NetCDF Context manager (*Saver*) directly.
- The save process will stream the data payload to the file using dask, enabling large data payloads to be saved and maintaining the 'lazy' status of the cube's data payload, unless the netcdf_format is explicitly specified to be 'NETCDF3' or 'NETCDF3_CLASSIC'.

Args:

- **cube (*iris.cube.Cube* or *iris.cube.CubeList*):** A *iris.cube.Cube*, *iris.cube.CubeList* or other iterable of cubes to be saved to a netCDF file.
- **filename (string):** Name of the netCDF file to save the cube(s).

Kwargs:

- **netcdf_format (string):** Underlying netCDF file format, one of 'NETCDF4', 'NETCDF4_CLASSIC', 'NETCDF3_CLASSIC' or 'NETCDF3_64BIT'. Default is 'NETCDF4' format.
- **local_keys (iterable of strings):** An interable of cube attribute keys. Any cube attributes with matching keys will become attributes on the data variable rather than global attributes.
- **unlimited_dimensions (iterable of strings and/or**

  ***iris.coords.Coord* objects):** List of coordinate names (or coordinate objects) corresponding to coordinate dimensions of *cube* to save with the NetCDF dimension variable length 'UNLIMITED'. By default, no unlimited dimensions are saved. Only the 'NETCDF4' format supports multiple 'UNLIMITED' dimensions.
- **zlib (bool):** If *True*, the data will be compressed in the netCDF file using gzip compression (default *False*).
- **complevel (int):** An integer between 1 and 9 describing the level of compression desired (default 4). Ignored if *zlib=False*.
- **shuffle (bool):** If *True*, the HDF5 shuffle filter will be applied before compressing the data (default *True*). This significantly improves compression. Ignored if *zlib=False*.
- **fletcher32 (bool):** If *True*, the Fletcher32 HDF5 checksum algorithm is activated to detect errors. Default *False*.
- **contiguous (bool):** If *True*, the variable data is stored contiguously on disk. Default *False*. Setting to *True* for a variable with an unlimited dimension will trigger an error.
- **chunksizes (tuple of int):** Used to manually specify the HDF5 chunksizes for each dimension of the variable. A detailed discussion of HDF chunking and I/O performance is available here: https://www.unidata.ucar.edu/software/netcdf/documentation/NUG/netcdf_perf_chunking.html. Basically, you want the chunk size for each dimension to match as closely as possible the size of

the data block that users will read from the file. *chunksizes* cannot be set if *contiguous=True*.

- **endian (string):** Used to control whether the data is stored in little or big endian format on disk. Possible values are 'little', 'big' or 'native' (default). The library will automatically handle endian conversions when the data is read, but if the data is always going to be read on a computer with the opposite format as the one used to create the file, there may be some performance advantage to be gained by setting the endian-ness.
- **least_significant_digit (int):** If *least_significant_digit* is specified, variable data will be truncated (quantized). In conjunction with *zlib=True* this produces 'lossy', but significantly more efficient compression. For example, if *least_significant_digit=1*, data will be quantized using *numpy.around(scale*data)/scale*, where *scale = 2**bits*, and *bits* is determined so that a precision of 0.1 is retained (in this case *bits=4*). From http://www.esrl.noaa.gov/psd/data/gridded/conventions/cdc_netcdf_standard.shtml: "least_significant_digit – power of ten of the smallest decimal place in unpacked data that is a reliable value". Default is *None*, or no quantization, or 'lossless' compression.
- **packing (type or string or dict or list): A numpy integer datatype** (signed or unsigned) or a string that describes a numpy integer dtype (i.e. 'i2', 'short', 'u4') or a dict of packing parameters as described below or an iterable of such types, strings, or dicts. This provides support for netCDF data packing as described in https://www.unidata.ucar.edu/software/netcdf/documentation/NUG/best_practices.html#bp_Packed-Data-Values If this argument is a type (or type string), appropriate values of scale_factor and add_offset will be automatically calculated based on *cube.data* and possible masking. For more control, pass a dict with one or more of the following keys: *dtype* (required), *scale_factor* and *add_offset*. Note that automatic calculation of packing parameters will trigger loading of lazy data; set them manually using a dict to avoid this. The default is *None*, in which case the datatype is determined from the cube and no packing will occur. If this argument is a list it must have the same number of elements as *cube* if *cube* is a *:class:`iris.cube.CubeList*, or one element, and each element of this argument will be applied to each cube separately.
- **fill_value (numeric or list):** The value to use for the *_FillValue* attribute on the netCDF variable. If *packing* is specified the value of *fill_value* should be in the domain of the packed data. If this argument is a list it must have the same number of elements as *cube* if *cube* is a *:class:`iris.cube.CubeList*, or a single element, and each element of this argument will be applied to each cube separately.

Returns None.

Note: The *zlib*, *complevel*, *shuffle*, *fletcher32*, *contiguous*, *chunksizes* and *endian* keywords are silently ignored for netCDF 3 files that do not use HDF5.

See also:

NetCDF Context manager (`Saver`).

Provide a simple CF name to CF coordinate mapping.

class iris.fileformats.netcdf.**CFNameCoordMap**
Provide a simple CF name to CF coordinate mapping.

append(*name*, *coord*)
Append the given name and coordinate pair to the mapping.

Args:
- **name:** CF name of the associated coordinate.
- **coord:** The coordinate of the associated CF name.

Returns None.

coord(*name*)
Return the coordinate, given a CF name.

Args:
- **name:** CF name of the associated coordinate.

Returns CF name.

name(*coord*)
Return the CF name, given a coordinate

Args:
- **coord:** The coordinate of the associated CF name.

Returns Coordinate.

property coords
Return all the coordinates.

property names
Return all the CF names.

A reference to the data payload of a single NetCDF file variable.

class iris.fileformats.netcdf.**NetCDFDataProxy**(*shape*, *dtype*, *path*, *variable_name*, *fill_value*)
A reference to the data payload of a single NetCDF file variable.

dtype

fill_value

property ndim

path

shape

variable_name

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

class iris.fileformats.netcdf.**OrderedAddableList**(**args*, ***kwargs*)

add(*msg*)

append(*object*, /)
Append object to the end of the list.

clear()
Remove all items from list.

copy()
Return a shallow copy of the list.

count(*value*, /)
Return number of occurrences of value.

extend(*iterable*, /)
Extend list by appending elements from the iterable.

index(*value*, *start=0*, *stop=9223372036854775807*, /)
Return first index of value.

Raises ValueError if the value is not present.

insert(*index*, *object*, /)
Insert object before index.

pop(*index=- 1*, /)
Remove and return item at index (default last).

Raises IndexError if list is empty or index is out of range.

remove(*value*, /)
Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse()
Reverse *IN PLACE*.

sort(*, *key=None*, *reverse=False*)
Sort the list in ascending order and return None.

The sort is in-place (i.e. the list itself is modified) and stable (i.e. the order of two equal elements is maintained).

If a key function is given, apply it once to each list item and sort them, ascending or descending, according to their function values.

The reverse flag can be set to sort in descending order.

A manager for saving netcdf files.

class iris.fileformats.netcdf.**Saver**(*filename*, *netcdf_format*)
A manager for saving netcdf files.

Args:
- **filename (string):** Name of the netCDF file to save the cube.
- **netcdf_format (string):** Underlying netCDF file format, one of 'NETCDF4', 'NETCDF4_CLASSIC', 'NETCDF3_CLASSIC' or 'NETCDF3_64BIT'. Default is 'NETCDF4' format.

Returns None.

For example:

```
# Initialise Manager for saving
with Saver(filename, netcdf_format) as sman:
    # Iterate through the cubelist.
    for cube in cubes:
        sman.write(cube)
```

__exit__(*type*, *value*, *traceback*)
Flush any buffered data to the CF-netCDF file before closing.

static cf_valid_var_name(*var_name*)
Return a valid CF var_name given a potentially invalid name.

Args:
- **var_name (str):** The var_name to normalise

Returns A var_name suitable for passing through for variable creation.

static check_attribute_compliance(*container*, *data*)

update_global_attributes(*attributes=None*, ***kwargs*)
Update the CF global attributes based on the provided iterable/dictionary and/or keyword arguments.

Args:
- **attributes (dict or iterable of key, value pairs):** CF global attributes to be updated.

write(*cube*, *local_keys=None*, *unlimited_dimensions=None*, *zlib=False*, *complevel=4*, *shuffle=True*, *fletcher32=False*, *contiguous=False*, *chunksizes=None*, *endian='native'*, *least_significant_digit=None*, *packing=None*, *fill_value=None*)
Wrapper for saving cubes to a NetCDF file.

Args:
- **cube (*iris.cube.Cube*):** A *iris.cube.Cube* to be saved to a netCDF file.

Kwargs:
- **local_keys (iterable of strings):** An interable of cube attribute keys. Any cube attributes with matching keys will become attributes on the data variable rather than global attributes.
- **unlimited_dimensions (iterable of strings and/or *iris.coords.Coord* objects):** List of coordinate names (or coordinate objects) corresponding to coordinate dimensions of *cube* to save with the NetCDF dimension variable length 'UNLIMITED'. By default, no unlimited dimensions are saved. Only the 'NETCDF4' format supports multiple 'UNLIMITED' dimensions.
- **zlib (bool):** If *True*, the data will be compressed in the netCDF file using gzip compression (default *False*).
- **complevel (int):** An integer between 1 and 9 describing the level of compression desired (default 4). Ignored if *zlib=False*.
- **shuffle (bool):** If *True*, the HDF5 shuffle filter will be applied before compressing the data (default *True*). This significantly improves compression. Ignored if *zlib=False*.
- **fletcher32 (bool):** If *True*, the Fletcher32 HDF5 checksum algorithm is activated to detect errors. Default *False*.

- **contiguous (bool):** If *True*, the variable data is stored contiguously on disk. Default *False*. Setting to *True* for a variable with an unlimited dimension will trigger an error.
- **chunksizes (tuple of int):** Used to manually specify the HDF5 chunksizes for each dimension of the variable. A detailed discussion of HDF chunking and I/O performance is available here: https://www.unidata.ucar.edu/software/netcdf/documentation/NUG/netcdf_perf_chunking.html. Basically, you want the chunk size for each dimension to match as closely as possible the size of the data block that users will read from the file. *chunksizes* cannot be set if *contiguous=True*.
- **endian (string):** Used to control whether the data is stored in little or big endian format on disk. Possible values are 'little', 'big' or 'native' (default). The library will automatically handle endian conversions when the data is read, but if the data is always going to be read on a computer with the opposite format as the one used to create the file, there may be some performance advantage to be gained by setting the endian-ness.
- **least_significant_digit (int):** If *least_significant_digit* is specified, variable data will be truncated (quantized). In conjunction with *zlib=True* this produces 'lossy', but significantly more efficient compression. For example, if *least_significant_digit=1*, data will be quantized using *numpy.around(scale*data)/scale*, where *scale = 2**bits*, and *bits* is determined so that a precision of 0.1 is retained (in this case *bits=4*). From http://www.esrl.noaa.gov/psd/data/gridded/conventions/cdc_netcdf_standard.shtml: "least_significant_digit – power of ten of the smallest decimal place in unpacked data that is a reliable value". Default is *None*, or no quantization, or 'lossless' compression.
- **packing (type or string or dict or list): A numpy integer datatype** (signed or unsigned) or a string that describes a numpy integer dtype(i.e. 'i2', 'short', 'u4') or a dict of packing parameters as described below. This provides support for netCDF data packing as described in https://www.unidata.ucar.edu/software/netcdf/documentation/NUG/best_practices.html#bp_Packed-Data-Values If this argument is a type (or type string), appropriate values of scale_factor and add_offset will be automatically calculated based on *cube.data* and possible masking. For more control, pass a dict with one or more of the following keys: *dtype* (required), *scale_factor* and *add_offset*. Note that automatic calculation of packing parameters will trigger loading of lazy data; set them manually using a dict to avoid this. The default is *None*, in which case the datatype is determined from the cube and no packing will occur.
- **fill_value:** The value to use for the *_FillValue* attribute on the netCDF variable. If *packing* is specified the value of *fill_value* should be in the domain of the packed data.

Returns None.

Note: The *zlib*, *complevel*, *shuffle*, *fletcher32*, *contiguous*, *chunksizes* and *endian* keywords are silently ignored for netCDF 3 files that do not use HDF5.

Base class for warning categories.

class iris.fileformats.netcdf.**UnknownCellMethodWarning**

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

args

27.11.7 iris.fileformats.nimrod

Provides NIMROD file format capabilities.

In this module:

- *load_cubes*
- *NimrodField*

iris.fileformats.nimrod.**load_cubes**(*filenames*, *callback=None*)
Loads cubes from a list of NIMROD filenames.

Args:

- filenames - list of NIMROD filenames to load

Kwargs:

- **callback - a function which can be passed on to** *iris.io.run_callback()*

Note: The resultant cubes may not be in the same order as in the files.

A data field from a NIMROD file.

Capable of converting itself into a *Cube*

References: Met Office (2003): Met Office Rain Radar Data from the NIMROD System. NCAS British Atmospheric Data Centre, date of citation. http://catalogue.ceda.ac.uk/uuid/82adec1f896af6169112d09cc1174499

class iris.fileformats.nimrod.**NimrodField**(*from_file=None*)
Create a NimrodField object and optionally read from an open file.

Example:

```
with open("nimrod_file", "rb") as infile:
    field = NimrodField(infile)
```

read(*infile*)
Read the next field from the given file object.

27.11.8 iris.fileformats.nimrod_load_rules

Rules for converting NIMROD fields into cubes.

In this module:

- *run*

iris.fileformats.nimrod_load_rules.**run**(*field*, *handle_metadata_errors=True*)
Convert a NIMROD field to an Iris cube.

:param * field - a *NimrodField*: :param * handle_metadata_errors - Set to False to omit handling of known meta-data deficiencies: in Nimrod-format data

Returns

- A new *Cube*, created from the NimrodField.

27.11.9 iris.fileformats.pp

Provides UK Met Office Post Process (PP) format specific capabilities.

In this module:

- *load*
- *save*
- *load_cubes*
- *PPField*
- *as_fields*
- *load_pairs_from_fields*
- *save_pairs_from_cube*
- *save_fields*
- *STASH*
- *EARTH_RADIUS*

iris.fileformats.pp.**load**(*filename*, *read_data=False*, *little_ended=False*)
Return an iterator of PPFields given a filename.

Args:

- filename - string of the filename to load.

Kwargs:

- **read_data - boolean** Flag whether or not the data should be read, if False an empty data manager will be provided which can subsequently load the data on demand. Default False.
- **little_ended - boolean** If True, file contains all little-ended words (header and data).

To iterate through all of the fields in a pp file:

```
for field in iris.fileformats.pp.load(filename):
    print(field)
```

iris.fileformats.pp.**save**(*cube*, *target*, *append=False*, *field_coords=None*)
Use the PP saving rules (and any user rules) to save a cube to a PP file.

:param * cube - A *iris.cube.Cube*: :param * target - A filename or open file handle.:

Kwargs:

- **append - Whether to start a new file afresh or add the cube(s)** to the end of the file. Only applicable when target is a filename, not a file handle. Default is False.
- **field_coords - list of 2 coords or coord names which are to be used** for reducing the given cube into 2d slices, which will ultimately determine the x and y coordinates of the resulting fields. If None, the final two dimensions are chosen for slicing.

See also *iris.io.save()*. Note that *iris.save()* is the preferred method of saving. This allows a *iris.cube.CubeList* or a sequence of cubes to be saved to a PP file.

---

iris.fileformats.pp.**load_cubes**(*filenames*, *callback=None*, *constraints=None*)
Loads cubes from a list of pp filenames.

Args:

- filenames - list of pp filenames to load

Kwargs:

- constraints - a list of Iris constraints
- **callback - a function which can be passed on to** *iris.io.run_callback()*

Note: The resultant cubes may not be in the order that they are in the file (order is not preserved when there is a field with orography references)

A generic class for PP fields - not specific to a particular header release number.

A PPField instance can easily access the PP header "words" as attributes with some added useful capabilities:

```
for field in iris.fileformats.pp.load(filename):
    print(field.lbyr)
    print(field.lbuser)
    print(field.lbuser[0])
    print(field.lbtim)
    print(field.lbtim.ia)
    print(field.t1)
```

class iris.fileformats.pp.**PPField**(*header=None*)
A generic class for PP fields - not specific to a particular header release number.

A PPField instance can easily access the PP header "words" as attributes with some added useful capabilities:

```
for field in iris.fileformats.pp.load(filename):
    print(field.lbyr)
    print(field.lbuser)
    print(field.lbuser[0])
    print(field.lbtim)
    print(field.lbtim.ia)
    print(field.t1)
```

__getattr__(*key*)
This method supports deferred attribute creation, which offers a significant loading optimisation, particularly when not all attributes are referenced and therefore created on the instance.

When an 'ordinary' HEADER_DICT attribute is required, its associated header offset is used to lookup the data value/s from the combined header longs and floats data cache. The attribute is then set with this value/s on the instance. Thus future lookups for this attribute will be optimised, avoiding the __getattr__ lookup mechanism again.

When a 'special' HEADER_DICT attribute (leading underscore) is required, its associated 'ordinary' (no leading underscore) header offset is used to lookup the data value/s from the combined header longs and floats data cache. The 'ordinary' attribute is then set with this value/s on the instance. This is required as 'special' attributes have supporting property convenience functionality base on the attribute value e.g. see 'lbpack' and 'lbtim'. Note that, for 'special' attributes the interface is via the 'ordinary' attribute but the underlying attribute value is stored within the 'special' attribute.

__repr__()
Return a string representation of the PP field.

coord_system()
Return a CoordSystem for this PPField.
Returns Currently, a *GeogCS* or *RotatedGeogCS*.

copy()
Returns a deep copy of this PPField.
Returns A copy instance of the *PPField*.

core_data()

save(*file_handle*)
Save the PPField to the given file object (typically created with open()).

```
# to append the field to a file
with open(filename, 'ab') as fh:
    a_pp_field.save(fh)

# to overwrite/create a file
with open(filename, 'wb') as fh:
    a_pp_field.save(fh)
```

Note: The fields which are automatically calculated are: 'lbext', 'lblrec' and 'lbuser[0]'. Some fields are not currently populated, these

are: ‘lbegin’, ‘lbnrec’, ‘lbuser[1]’.

---

time_unit(*time_unit*, *epoch='epoch'*)

property calendar
Return the calendar of the field.

property data
The numpy.ndarray representing the multidimensional data of the pp file

property lbcode

property lbpack

property lbproc

property lbtim

property stash
A stash property giving access to the associated STASH object, now supporting __eq__

abstract property t1

abstract property t2

property x_bounds

property y_bounds

---

iris.fileformats.pp.**as_fields**(*cube*, *field_coords=None*, *target=None*)
Use the PP saving rules (and any user rules) to convert a cube to an iterable of PP fields.

Args:

- **cube:** A *iris.cube.Cube*

Kwargs:

- **field_coords:** List of 2 coords or coord names which are to be used for reducing the given cube into 2d slices, which will ultimately determine the x and y coordinates of the resulting fields. If None, the final two dimensions are chosen for slicing.
- **target:** A filename or open file handle.

---

iris.fileformats.pp.**load_pairs_from_fields**(*pp_fields*)
Convert an iterable of PP fields into an iterable of tuples of (Cubes, PPField).

Args:

- **pp_fields:** An iterable of *iris.fileformats.pp.PPField*.

Returns An iterable of *iris.cube.Cube*s.

This capability can be used to filter out fields before they are passed to the load pipeline, and amend the cubes once they are created, using PP metadata conditions. Where this filtering removes a significant number of fields, the speed up to load can be significant:

```
>>> import iris
>>> from iris.fileformats.pp import load_pairs_from_fields
>>> filename = iris.sample_data_path('E1.2098.pp')
>>> filtered_fields = []
>>> for field in iris.fileformats.pp.load(filename):
...     if field.lbproc == 128:
...         filtered_fields.append(field)
>>> cube_field_pairs = load_pairs_from_fields(filtered_fields)
>>> for cube, field in cube_field_pairs:
...     cube.attributes['lbproc'] = field.lbproc
...     print(cube.attributes['lbproc'])
128
```

This capability can also be used to alter fields before they are passed to the load pipeline. Fields with out of specification header elements can be cleaned up this way and cubes created:

```
>>> filename = iris.sample_data_path('E1.2098.pp')
>>> cleaned_fields = list(iris.fileformats.pp.load(filename))
>>> for field in cleaned_fields:
...     if field.lbrel == 0:
...         field.lbrel = 3
>>> cubes_field_pairs = list(load_pairs_from_fields(cleaned_
↪fields))
```

---

iris.fileformats.pp.**save_pairs_from_cube**(*cube*, *field_coords=None*, *target=None*)

Use the PP saving rules to convert a cube or iterable of cubes to an iterable of (2D cube, PP field) pairs.

Args:

- **cube:** A *iris.cube.Cube*

Kwargs:

- **field_coords:** List of 2 coords or coord names which are to be used for reducing the given cube into 2d slices, which will ultimately determine the x and y coordinates of the resulting fields. If None, the final two dimensions are chosen for slicing.
- **target:** A filename or open file handle.

---

iris.fileformats.pp.**save_fields**(*fields*, *target*, *append=False*)

Save an iterable of PP fields to a PP file.

Args:

- **fields:** An iterable of PP fields.
- **target:** A filename or open file handle.

Kwargs:

- **append:** Whether to start a new file afresh or add the cube(s) to the end of the file. Only applicable when target is a filename, not a file handle. Default is False.

See also *iris.io.save()*.

A class to hold a single STASH code.

Create instances using:

```
>>> model = 1
>>> section = 2
>>> item = 3
>>> my_stash = iris.fileformats.pp.STASH(model, section, item)
```

Access the sub-components via:

```
>>> my_stash.model
1
>>> my_stash.section
2
>>> my_stash.item
3
```

String conversion results in the MSI format:

```
>>> print(iris.fileformats.pp.STASH(1, 16, 203))
m01s16i203
```

A stash object can be compared directly to its string representation: >>> iris.fileformats.pp.STASH(1, 0, 4) == 'm01s0i004' True

class iris.fileformats.pp.**STASH**(*model*, *section*, *item*)

Args:

- **model** A positive integer less than 100, or None.
- **section** A non-negative integer less than 100, or None.
- **item** A positive integer less than 1000, or None.

static __new__(*cls*, *model*, *section*, *item*)

Args:

- **model** A positive integer less than 100, or None.
- **section** A non-negative integer less than 100, or None.
- **item** A positive integer less than 1000, or None.

count(*value*, /)

Return number of occurrences of value.

static from_msi(*msi*)

Convert a STASH code MSI string to a STASH instance.

index(*value*, *start=0*, *stop=9223372036854775807*, /)

Return first index of value.

Raises ValueError if the value is not present.

lbuser3()

Return the lbuser[3] value that this stash represents.

lbuser6()
Return the lbuser[6] value that this stash represents.

property is_valid

item
Alias for field number 2

model
Alias for field number 0

section
Alias for field number 1

---

iris.fileformats.pp.**EARTH_RADIUS**
Convert a string or number to a floating point number, if possible.

27.11.10 iris.fileformats.pp_load_rules

In this module:

- *convert*

iris.fileformats.pp_load_rules.**convert**(*f*)
Converts a PP field into the corresponding items of Cube metadata.

Args:

- **f:** A *iris.fileformats.pp.PPField* object.

Returns A *iris.fileformats.rules.ConversionMetadata* object.

27.11.11 iris.fileformats.pp_save_rules

In this module:

- *verify*

iris.fileformats.pp_save_rules.**verify**(*cube*, *field*)

27.11.12 iris.fileformats.rules

Generalised mechanisms for metadata translation and cube construction.

In this module:

- *aux_factory*
- *has_aux_factory*
- *load_cubes*
- *load_pairs_from_fields*
- *scalar_cell_method*
- *scalar_coord*
- *vector_coord*

- *ConcreteReferenceTarget*
- *ConversionMetadata*
- *Factory*
- *Loader*
- *Reference*
- *ReferenceTarget*

iris.fileformats.rules.**aux_factory**(*cube*, *aux_factory_class*)
Return the class:~*iris.aux_factory.AuxCoordFactory* instance of the specified type from a cube.

---

iris.fileformats.rules.**has_aux_factory**(*cube*, *aux_factory_class*)
Try to find an class:~*iris.aux_factory.AuxCoordFactory* instance of the specified type on the cube.

---

iris.fileformats.rules.**load_cubes**(*filenames*, *user_callback*, *loader*, *filter_function=None*)

---

iris.fileformats.rules.**load_pairs_from_fields**(*fields*, *converter*)
Convert an iterable of fields into an iterable of Cubes using the provided convertor.

Args:

- **fields:** An iterable of fields.
- **convertor:** An Iris convertor function, suitable for use with the supplied fields. See the description in *iris.fileformats.rules.Loader*.

Returns An iterable of (*iris.cube.Cube*, field) pairs.

---

iris.fileformats.rules.**scalar_cell_method**(*cube*, *method*, *coord_name*)
Try to find the given type of cell method over a single coord with the given name.

---

iris.fileformats.rules.**scalar_coord**(*cube*, *coord_name*)
Try to find a single-valued coord with the given name.

---

iris.fileformats.rules.**vector_coord**(*cube*, *coord_name*)
Try to find a one-dimensional, multi-valued coord with the given name.

Everything you need to make a real Cube for a named reference.

class iris.fileformats.rules.**ConcreteReferenceTarget**(*name*, *transform=None*)
Everything you need to make a real Cube for a named reference.

add_cube(*cube*)

as_cube()

name
The name used to connect references with references.

transform
An optional transformation to apply to the cubes.

ConversionMetadata(factories, references, standard_name, long_name, units, attributes, cell_methods, dim_coords_and_dims, aux_coords_and_dims)

class iris.fileformats.rules.**ConversionMetadata**(*_cls*, *factories*, *references*, *standard_name*, *long_name*, *units*, *attributes*, *cell_methods*, *dim_coords_and_dims*, *aux_coords_and_dims*)

Create new instance of ConversionMetadata(factories, references, standard_name, long_name, units, attributes, cell_methods, dim_coords_and_dims, aux_coords_and_dims)

count(*value*, */*)
Return number of occurrences of value.

index(*value*, *start=0*, *stop=9223372036854775807*, */*)
Return first index of value.

Raises ValueError if the value is not present.

attributes
Alias for field number 5

aux_coords_and_dims
Alias for field number 8

cell_methods
Alias for field number 6

dim_coords_and_dims
Alias for field number 7

factories
Alias for field number 0

long_name
Alias for field number 3

references
Alias for field number 1

standard_name
Alias for field number 2

units
Alias for field number 4

Factory(factory_class, args)

class iris.fileformats.rules.**Factory**(*_cls*, *factory_class*, *args*)

Create new instance of Factory(factory_class, args)

count(*value*, /)

Return number of occurrences of value.

index(*value*, *start=0*, *stop=9223372036854775807*, /)

Return first index of value.

Raises ValueError if the value is not present.

args

Alias for field number 1

factory_class

Alias for field number 0

Loader(field_generator, field_generator_kwargs, converter)

class iris.fileformats.rules.**Loader**(*field_generator*, *field_generator_kwargs*, *converter*)

Create a definition of a field-based Cube loader.

Args:

- **field_generator** A callable that accepts a filename as its first argument and returns an iterable of field objects.
- **field_generator_kwargs** Additional arguments to be passed to the field_generator.
- **converter** A callable that converts a field object into a Cube.

static **__new__**(*cls*, *field_generator*, *field_generator_kwargs*, *converter*)

Create a definition of a field-based Cube loader.

Args:

- **field_generator** A callable that accepts a filename as its first argument and returns an iterable of field objects.
- **field_generator_kwargs** Additional arguments to be passed to the field_generator.
- **converter** A callable that converts a field object into a Cube.

count(*value*, /)

Return number of occurrences of value.

index(*value*, *start=0*, *stop=9223372036854775807*, /)

Return first index of value.

Raises ValueError if the value is not present.

converter

Alias for field number 2

field_generator

Alias for field number 0

field_generator_kwargs
Alias for field number 1

Convenience class for creating "immutable", hashable, and ordered classes.

Instance identity is defined by the specific list of attribute names declared in the abstract attribute "_names". Subclasses must declare the attribute "_names" as an iterable containing the names of all the attributes relevant to equality/hash-value/ordering.

Initial values should be set by using :: self._init(self, value1, value2, ..)

Note: It's the responsibility of the subclass to ensure that the values of its attributes are themselves hashable.

---

class iris.fileformats.rules.**Reference**(*name*)

ReferenceTarget(name, transform)

class iris.fileformats.rules.**ReferenceTarget**(*_cls*, *name*, *transform*)
Create new instance of ReferenceTarget(name, transform)

count(*value*, /)
Return number of occurrences of value.

index(*value*, *start=0*, *stop=9223372036854775807*, /)
Return first index of value.

Raises ValueError if the value is not present.

name
Alias for field number 0

transform
Alias for field number 1

27.11.13 iris.fileformats.um

Provides iris loading support for UM Fieldsfile-like file types, and PP.

At present, the only UM file types supported are true FieldsFiles and LBCs. Other types of UM file may fail to load correctly (or at all).

In this module:

- *um_to_pp*
- *load_cubes*
- *load_cubes_32bit_ieee*
- *structured_um_loading*
- *FieldCollation*

iris.fileformats.um.**um_to_pp**(*filename*, *read_data=False*, *word_depth=None*)
Extract individual PPFields from within a UM Fieldsfile-like file.

Returns an iterator over the fields contained within the FieldsFile, returned as *iris.fileformats.pp.PPField* instances.

Args:

- **filename (string):** Specify the name of the FieldsFile.

Kwargs:

- **read_data (boolean):** Specify whether to read the associated PPField data within the FieldsFile. Default value is False.

Returns Iteration of *iris.fileformats.pp.PPField*.

For example:

```
>>> for field in um.um_to_pp(filename):
...     print(field)
```

---

iris.fileformats.um.**load_cubes**(*filenames*, *callback*, *constraints=None*, *_loader_kwargs=None*)
Loads cubes from filenames of UM fieldsfile-like files.

Args:

- filenames - list of filenames to load

Kwargs:

- **callback - a function which can be passed on to** *iris.io.run_callback()*

Note: The resultant cubes may not be in the order that they are in the file (order is not preserved when there is a field with orography references).

---

iris.fileformats.um.**load_cubes_32bit_ieee**(*filenames*, *callback*, *constraints=None*)
Loads cubes from filenames of 32bit ieee converted UM fieldsfile-like files.

See also:

load_cubes() for keyword details

---

iris.fileformats.um.**structured_um_loading**()
Load cubes from structured UM Fieldsfile and PP files.

"Structured" loading is a streamlined, fast load operation, to be used **only** on fieldsfiles or PP files whose fields repeat regularly over the same vertical levels and times (see full details below).

This method is a context manager which enables an alternative loading mechanism for 'structured' UM files, providing much faster load times. Within the scope of the context manager, this affects all standard Iris load functions (*load()*, *load_cube()*,

load_cubes() and *load_raw()*), when loading from UM format files (PP or fields-files).

For example:

```
>>> import iris
>>> filepath = iris.sample_data_path('uk_hires.pp')
>>> from iris.fileformats.um import structured_um_loading
>>> with structured_um_loading():
...     cube = iris.load_cube(filepath, 'air_potential_
↪temperature')
...
>>> cube
<iris 'Cube' of air_potential_temperature / (K) (time: 3; model_
↪level_number: 7; grid_latitude: 204; grid_longitude: 187)>
```

The results from this are normally equivalent to those generated by *iris.load()*, but the operation is substantially faster for input which is structured.

For calls other than *load_raw()*, the resulting cubes are concatenated over all the input files, so there is normally just one output cube per phenomenon.

However, actual loaded results are somewhat different from non-structured loads in many cases, and in a variety of ways. Most commonly, dimension ordering and the choice of dimension coordinates are often different.

Use of load callbacks:

> When a user callback function is used with structured-loading, it is called in a somewhat different way than in a 'normal' load : The callback is called once for each basic *structured* cube loaded, which is normally the whole of one phenomenon from a single input file. In particular, the callback's "field" argument is a *FieldCollation*, from which "field.fields" gives a *list* of PPFields from which that cube was built, and the properties "field.load_filepath" and "field.load_file_indices" reference the original file locations of the cube data. The code required is therefore different from a 'normal' callback. For an example of this, see this example in the Iris test code.

Notes on applicability:

> For results to be **correct and reliable**, the input files must conform to the following requirements :
>
> - the file must contain fields for all possible combinations of the vertical levels and time points found in the file.
> - the fields must occur in a regular repeating order within the file, within the fields of each phenomenon.
>
>   For example: a sequence of fields for NV vertical levels, repeated for NP different forecast periods, repeated for NT different forecast times.
>
> - all other metadata must be identical across all fields of the same phenomenon.
>
> Each group of fields with the same values of LBUSER4, LBUSER7 and LBPROC is identified as a separate phenomenon: These groups are processed independently and returned as separate result cubes. The need for

a regular sequence of fields applies separately to the fields of each phenomenon, such that different phenomena may have different field structures, and can be interleaved in any way at all.

Note: At present, fields with different values of 'LBUSER5' (pseudo-level) are *also* treated internally as different phenomena, yielding a raw cube per level. The effects of this are not normally noticed, as the resulting multiple raw cubes merge together again in a 'normal' load. However, it is not an ideal solution as operation is less efficient (in particular, slower) : it is done to avoid a limitation in the underlying code which would otherwise load data on pseudo-levels incorrectly. In future, this may be corrected.

Known current shortcomings:

- orography fields may be returned with extra dimensions, e.g. time, where multiple fields exist in an input file.
- if some input files contain a *single* coordinate value while others contain *multiple* values, these will not be merged into a single cube over all input files : Instead, the single- and multiple-valued sets will typically produce two separate cubes with overlapping coordinates.
  - this can be worked around by loading files individually, or with *load_raw()*, and merging/concatenating explicitly.

Note: The resulting time-related coordinates ('time', 'forecast_time' and 'forecast_period') may be mapped to shared cube dimensions and in some cases can also be multidimensional. However, the vertical level information *must* have a simple one-dimensional structure, independent of the time points, otherwise an error will be raised.

Note: Where input data does *not* have a fully regular arrangement, the corresponding result cube will have a single anonymous extra dimension which indexes over all the input fields.

This can happen if, for example, some fields are missing; or have slightly different metadata; or appear out of order in the file.

Warning: Restrictions and limitations:

Any non-regular metadata variation in the input should be strictly avoided, as not all irregularities are detected, which can cause erroneous results.

Various field header words which can in some cases vary are assumed to have a constant value throughout a given phenomenon. This is **not** checked, and can lead to erroneous results if it is not the case. Header elements of potential concern include LBTIM, LBCODE, LBVC and LBRSVD4 (ensemble number).

An object representing a group of UM fields with array structure that can be vectorized into a single cube.

For example:

Suppose we have a set of 28 fields repeating over 7 vertical levels for each of 4 different data times. If a BasicFieldCollation is created to contain these, it can identify that this is a 4*7 regular array structure.

This BasicFieldCollation will then have the following properties:

- **within 'element_arrays_and_dims' :** Element 'blev' have the array shape (7,) and dims of (1,). Elements 't1' and 't2' have shape (4,) and dims (0,). The other elements (lbft, lbrsvd4 and lbuser5) all have scalar array values and dims=None.

Note: If no array structure is found, the element values are all either scalar or full-length 1-D vectors.

---

class iris.fileformats.um.**FieldCollation**(*fields*, *filepath*)

Args:

- **fields (iterable of *iris.fileformats.pp.PPField*):** The fields in the collation.
- **filepath (string):** The path of the file the collation is loaded from.

core_data()

property bmdi

property data

property data_field_indices

Field indices of the contained PPFields in the input file.

This records the original file location of the individual data fields contained, within the input datafile.

Returns An integer array of shape *self.vector_dims_shape*.

property data_filepath

property data_proxy

property element_arrays_and_dims

Value arrays for vector metadata elements.

A dictionary mapping element_name: (value_array, dims).

The arrays are reduced to their minimum dimensions. A scalar array has an associated 'dims' of None (instead of an empty tuple).

property fields

property realised_dtype

property vector_dims_shape

The shape of the array structure.

27.11.14 iris.fileformats.um_cf_map

Provides UM/CF phenomenon translations.

In this module:

- *CFName*

CFName(standard_name, long_name, units)

class iris.fileformats.um_cf_map.**CFName**(*_cls*, *standard_name*, *long_name*, *units*)

Create new instance of CFName(standard_name, long_name, units)

count(*value*, */*)
Return number of occurrences of value.

index(*value*, *start=0*, *stop=9223372036854775807*, */*)
Return first index of value.

Raises ValueError if the value is not present.

long_name
Alias for field number 1

standard_name
Alias for field number 0

units
Alias for field number 2

A package for converting cubes to and from specific file formats.

In this module:

- *FORMAT_AGENT*

iris.fileformats.**FORMAT_AGENT**
The FORMAT_AGENT is responsible for identifying the format of a given URI. New formats can be added with the **add_spec** method.

27.12 iris.io

27.12.1 iris.io.format_picker

A module to provide convenient file format identification through a combination of filename extension and file based *magic* numbers.

To manage a collection of FormatSpecifications for loading:

```
import iris.io.format_picker as fp
import matplotlib.pyplot as plt
fagent = fp.FormatAgent()
png_spec = fp.FormatSpecification('PNG image', fp.MagicNumber(8),
                                  0x89504E470D0A1A0A,
                                  handler=lambda filename: plt.
↪imread(filename),
                                  priority=5
```

(continues on next page)

(continued from previous page)

```
                                    )
fagent.add_spec(png_spec)
```

To identify a specific format from a file:

```
with open(png_filename, 'rb') as png_fh:
    handling_spec = fagent.get_spec(png_filename, png_fh)
```

In the example, handling_spec will now be the png_spec previously added to the agent.

Now that a specification has been found, if a handler has been given with the specification, then the file can be handled:

```
handler = handling_spec.handler
if handler is None:
   raise ValueError('File cannot be handled.')
else:
   result = handler(filename)
```

The calling sequence of handler is dependent on the function given in the original specification and can be customised to your project's needs.

In this module:

- *FileElement*
- *FileExtension*
- *FormatAgent*
- *FormatSpecification*
- *LeadingLine*
- *MagicNumber*
- *UriProtocol*

Represents a specific aspect of a FileFormat which can be identified using the given element getter function.

class iris.io.format_picker.**FileElement**(*requires_fh=True*)

Constructs a new file element, which may require a file buffer.

Kwargs:

- requires_fh - Whether this FileElement needs a file buffer.

get_element(*basename*, *file_handle*)

Called when identifying the element of a file that this FileElement is representing.

A *FileElement* that returns the extension from the filename.

class iris.io.format_picker.**FileExtension**(*requires_fh=True*)

Constructs a new file element, which may require a file buffer.

Kwargs:

- requires_fh - Whether this FileElement needs a file buffer.

get_element(*basename*, *file_handle*)
Called when identifying the element of a file that this FileElement is representing.

The FormatAgent class is the containing object which is responsible for identifying the format of a given file by interrogating its children FormatSpecification instances.

Typically a FormatAgent will be created empty and then extended with the *FormatAgent.add_spec()* method:

```
agent = FormatAgent()
agent.add_spec(NetCDF_specification)
```

Less commonly, this can also be written:

```
agent = FormatAgent(NetCDF_specification)
```

class iris.io.format_picker.**FormatAgent**(*format_specs=None*)

add_spec(*format_spec*)
Add a FormatSpecification instance to this agent for format consideration.

get_spec(*basename*, *buffer_obj*)
Pick the first FormatSpecification which can handle the given filename and file/buffer object.

Note: `buffer_obj` may be `None` when a seekable file handle is not feasible (such as over the http protocol). In these cases only the format specifications which do not require a file handle are tested.

Provides the base class for file type definition.

Every FormatSpecification instance has a name which can be accessed with the *FormatSpecification.name* property and a FileElement, such as filename extension or 32-bit magic number, with an associated value for format identification.

class iris.io.format_picker.**FormatSpecification**(*format_name*, *file_element*, *file_element_value*, *handler=None*, *priority=0*, *constraint_aware_handler=False*)
Constructs a new FormatSpecification given the format_name and particular FileElements

Args:

- format_name - string name of fileformat being described
- file_element - FileElement instance of the element which identifies this FormatSpecification
- file_element_value - The value that the file_element should take if a file matches this FormatSpecification

Kwargs:

- **handler - function which will be called when the specification has been identified and is required to h**
  If None, then the file can still be identified but no handling can be done.
- priority - Integer giving a priority for considering this specification where higher priority means sooner consideration.

property file_element

property file_element_value

property handler
The handler function of this FileFormat. (Read only)

property name
The name of this FileFormat. (Read only)

A *FileElement* that returns the first line from the file.

class iris.io.format_picker.**LeadingLine**(*requires_fh=True*)
Constructs a new file element, which may require a file buffer.

Kwargs:

- requires_fh - Whether this FileElement needs a file buffer.

get_element(*basename*, *file_handle*)
Called when identifying the element of a file that this FileElement is representing.

A *FileElement* that returns a byte sequence in the file.

class iris.io.format_picker.**MagicNumber**(*num_bytes*, *offset=None*)
A *FileElement* that returns a byte sequence in the file.

get_element(*basename*, *file_handle*)
Called when identifying the element of a file that this FileElement is representing.

len_formats = {4: '>L', 8: '>Q'}

A *FileElement* that returns the "scheme" and "part" from a URI, using *decode_uri()*.

class iris.io.format_picker.**UriProtocol**
A *FileElement* that returns the "scheme" and "part" from a URI, using *decode_uri()*.

get_element(*basename*, *file_handle*)
Called when identifying the element of a file that this FileElement is representing.

Provides an interface to manage URI scheme support in iris.

In this module:

- *add_saver*
- *decode_uri*

- *expand_filespecs*
- *find_saver*
- *load_files*
- *load_http*
- *run_callback*
- *save*

iris.io.**add_saver**(*file_extension*, *new_saver*)
Add a custom saver to the Iris session.

Args:

- file_extension: A string such as "pp" or "my_format".
- new_saver: A function of the form `my_saver(cube, target)`.

See also *iris.io.save()*

---

iris.io.**decode_uri**(*uri*, *default='file'*)
Decodes a single URI into scheme and scheme-specific parts.

In addition to well-formed URIs, it also supports bare file paths. Both Windows and UNIX style paths are accepted.

Examples

```
>>> from iris.io import decode_uri
>>> print(decode_uri('http://www.thing.com:8080/resource?id=a:b'))
('http', '//www.thing.com:8080/resource?id=a:b')
```

```
>>> print(decode_uri('file:///data/local/dataZoo/...'))
('file', '///data/local/dataZoo/...')
```

```
>>> print(decode_uri('/data/local/dataZoo/...'))
('file', '/data/local/dataZoo/...')
```

```
>>> print(decode_uri('file:///C:\data\local\dataZoo\...'))
('file', '///C:\\data\\local\\dataZoo\\...')
```

```
>>> print(decode_uri('C:\data\local\dataZoo\...'))
('file', 'C:\\data\\local\\dataZoo\\...')
```

```
>>> print(decode_uri('dataZoo/...'))
('file', 'dataZoo/...')
```

---

iris.io.**expand_filespecs**(*file_specs*)
Find all matching file paths from a list of file-specs.

Args:

- **file_specs (iterable of string):** File paths which may contain '~' elements or wildcards.

Returns A well-ordered list of matching absolute file paths. If any of the file-specs match no existing files, an exception is raised.

---

iris.io.**find_saver**(*filespec*)

Find the saver function appropriate to the given filename or extension.

Parameters **"PP".** (** filespec - A string such as "my_file.pp" or*) –

Returns A save function or None. Save functions can be passed to *iris.io.save()*.

---

iris.io.**load_files**(*filenames*, *callback*, *constraints=None*)

Takes a list of filenames which may also be globs, and optionally a constraint set and a callback function, and returns a generator of Cubes from the given files.

Note: Typically, this function should not be called directly; instead, the intended interface for loading is *iris.load()*.

---

iris.io.**load_http**(*urls*, *callback*)

Takes a list of urls and a callback function, and returns a generator of Cubes from the given URLs.

Note: Typically, this function should not be called directly; instead, the intended interface for loading is *iris.load()*.

---

iris.io.**run_callback**(*callback*, *cube*, *field*, *filename*)

Runs the callback mechanism given the appropriate arguments.

Args:

- **callback:** A function to add metadata from the originating field and/or URI which obeys the following rules:
  1. Function signature must be: `(cube, field, filename)`.
  2. Modifies the given cube inplace, unless a new cube is returned by the function.
  3. If the cube is to be rejected the callback must raise an *iris.exceptions.IgnoreCubeException*.

Note: It is possible that this function returns None for certain callbacks, the caller of this function should handle this case.

---

iris.io.**save**(*source*, *target*, *saver=None*, ***kwargs*)

Save one or more Cubes to file (or other writeable).

Iris currently supports three file formats for saving, which it can recognise by filename extension:

- **netCDF - the Unidata network Common Data Format:**
  - see *iris.fileformats.netcdf.save()*
- **GRIB2 - the WMO GRIdded Binary data format:**
  - see `iris_grib.save_grib2()`.
- **PP - the Met Office UM Post Processing Format:**
  - see *iris.fileformats.pp.save()*

A custom saver can be provided to the function to write to a different file format.

Args:

- **source:** *iris.cube.Cube*, *iris.cube.CubeList* or sequence of cubes.
- **target:** A filename (or writeable, depending on file format). When given a filename or file, Iris can determine the file format.

Kwargs:

- **saver:** Optional. Specifies the file format to save. If omitted, Iris will attempt to determine the format.

  If a string, this is the recognised filename extension (where the actual filename may not have it). Otherwise the value is a saver function, of the form: `my_saver(cube, target)` plus any custom keywords. It is assumed that a saver will accept an `append` keyword if it's file format can handle multiple cubes. See also *iris.io.add_saver()*.

All other keywords are passed through to the saver function; see the relevant saver documentation for more information on keyword arguments.

Examples:

```
# Save a cube to PP
iris.save(my_cube, "myfile.pp")

# Save a cube list to a PP file, appending to the contents of the file
# if it already exists
iris.save(my_cube_list, "myfile.pp", append=True)

# Save a cube to netCDF, defaults to NETCDF4 file format
iris.save(my_cube, "myfile.nc")

# Save a cube list to netCDF, using the NETCDF3_CLASSIC storage option
iris.save(my_cube_list, "myfile.nc", netcdf_format="NETCDF3_CLASSIC")
```

Warning: Saving a cube whose data has been loaded lazily (if *cube.has_lazy_data()* returns *True*) to the same file it expects to load data from will cause both the data in-memory and the data on disk to be lost.

```
cube = iris.load_cube("somefile.nc")
# The next line causes data loss in 'somefile.nc' and the cube.
iris.save(cube, "somefile.nc")
```

In general, overwriting a file which is the source for any lazily loaded data can result in corruption. Users should proceed with caution when attempting to overwrite an existing file.

27.13 iris.iterate

Cube functions for iteration in step.

In this module:

- *izip*

iris.iterate.**izip**(**cubes*, ***kwargs*)
Return an iterator for iterating over a collection of cubes in step.

If the input cubes have dimensions for which there are no common coordinates, those dimensions will be treated as orthogonal. The resulting iterator will step through combinations of the associated coordinates.

Args:

- **cubes (*iris.cube.Cube*):** One or more *iris.cube.Cube* instances over which to iterate in step. Each cube should be provided as a separate argument e.g. `iris.iterate.izip(cube_a, cube_b, cube_c, ...)`.

Kwargs:

- **coords (string, coord or a list of strings/coords):** Coordinate names/coordinates of the desired subcubes (i.e. those that are not iterated over). They must all be orthogonal (i.e. point to different dimensions).
- **ordered (Boolean):** If True (default), the order of the coordinates in the resulting subcubes will match the order of the coordinates in the coords keyword argument. If False, the order of the coordinates will be preserved and will match that of the input cubes.

Returns An iterator over a collection of tuples that contain the resulting subcubes.

For example:

```
>>> e_content, e_density = iris.load_cubes(
...     iris.sample_data_path('space_weather.nc'),
...     ['total electron content', 'electron density'])
>>> for tslice, hslice in iris.iterate.izip(e_content, e_density,
...                                         coords=['grid_latitude',
...                                                 'grid_longitude
↪']):
...     pass
```

27.14 iris.palette

Load, configure and register color map palettes and initialise color map meta-data mappings.

In this module:

- *auto_palette*
- *cmap_norm*
- *is_brewer*
- *SymmetricNormalize*

iris.palette.**auto_palette**(*func*)

Decorator wrapper function to control the default behaviour of the matplotlib cmap and norm keyword arguments.

Args:

- **func (callable):** Callable function to be wrapped by the decorator.

Returns Closure wrapper function.

---

iris.palette.**cmap_norm**(*cube*)

Determine the default matplotlib.colors.LinearSegmentedColormap and *iris.palette.SymmetricNormalize* instances associated with the cube.

Args:

- **cube (*iris.cube.Cube*):** Source cube to generate default palette from.

Returns Tuple of matplotlib.colors.LinearSegmentedColormap and *iris.palette.SymmetricNormalize*

---

iris.palette.**is_brewer**(*cmap*)

Determine whether the color map is a Cynthia Brewer color map.

Args:

- **cmap:** The color map instance.

Returns Boolean.

Provides a symmetric normalization class around a given pivot point.

class iris.palette.**SymmetricNormalize**(*pivot*, **args*, ***kwargs*)

Provides a symmetric normalization class around a given pivot point.

__call__(*value*, *clip=None*)

Normalize *value* data in the `[vmin, vmax]` interval into the `[0.0, 1.0]` interval and return it.

Parameters

- **value** – Data to normalize.
- **clip** (*bool*) – If `None`, defaults to `self.clip` (which defaults to `False`).

Notes

If not already initialized, `self.vmin` and `self.vmax` are initialized using `self.autoscale_None(value)`.

autoscale(*A*)
Set *vmin*, *vmax* to min, max of *A*.

autoscale_None(*A*)
If vmin or vmax are not set, use the min/max of *A* to set them.

inverse(*value*)

static process_value(*value*)
Homogenize the input *value* for easy and efficient normalization.

value can be a scalar or sequence.

Returns

- **result** (*masked array*) – Masked array with the same shape as *value*.
- **is_scalar** (*bool*) – Whether *value* is a scalar.

Notes

Float dtypes are preserved; integer types with two bytes or smaller are converted to np.float32, and larger types are converted to np.float64. Preserving float32 when possible, and using in-place operations, greatly improves speed for large arrays.

scaled()
Return whether vmin and vmax are set.

property vmax

property vmin

27.15 iris.pandas

Provide conversion to and from Pandas data structures.

See also: http://pandas.pydata.org/

In this module:

- *as_cube*
- *as_data_frame*
- *as_series*

iris.pandas.**as_cube**(*pandas_array*, *copy=True*, *calendars=None*)
Convert a Pandas array into an Iris cube.

Parameters DataFrame. (** pandas_array - A Pandas Series or*) –

Kwargs:

- **copy - Whether to make a copy of the data.** Defaults to True.

- **calendars - A dict mapping a dimension to a calendar.** Required to convert datetime indices/columns.

Example usage:

```
as_cube(series, calendars={0: cf_units.CALENDAR_360_DAY})
as_cube(data_frame, calendars={1: cf_units.CALENDAR_GREGORIAN})
```

Note: This function will copy your data by default.

---

iris.pandas.**as_data_frame**(*cube*, *copy=True*)

Convert a 2D cube to a Pandas DataFrame.

Parameters **DataFrame.** (** cube - The cube to convert to a Pandas*) –

Kwargs:

- **copy - Whether to make a copy of the data.** Defaults to True. Must be True for masked data and some data types (see notes below).

Note: This function will copy your data by default. If you have a large array that cannot be copied, make sure it is not masked and use copy=False.

Note: Pandas will sometimes make a copy of the array, for example when creating from an int32 array. Iris will detect this and raise an exception if copy=False.

---

iris.pandas.**as_series**(*cube*, *copy=True*)

Convert a 1D cube to a Pandas Series.

Parameters **Series.** (** cube - The cube to convert to a Pandas*) –

Kwargs:

- **copy - Whether to make a copy of the data.** Defaults to True. Must be True for masked data.

Note: This function will copy your data by default. If you have a large array that cannot be copied, make sure it is not masked and use copy=False.

27.16 iris.plot

Iris-specific extensions to matplotlib, mimicking the `matplotlib.pyplot` interface.

See also: matplotlib.

In this module:

- *barbs*

- *citation*
- *contour*
- *contourf*
- *default_projection*
- *default_projection_extent*
- *orography_at_bounds*
- *orography_at_points*
- *outline*
- *pcolor*
- *pcolormesh*
- *plot*
- *points*
- *quiver*
- *scatter*
- *symbols*
- *PlotDefn*

iris.plot.**barbs**(*u_cube*, *v_cube*, **args*, ***kwargs*)

Draws a barb plot from two vector component cubes. Triangles, full-lines and half-lines represent increments of 50, 10 and 5 respectively.

Args:

- **u_cube, v_cube** [(*Cube*)] u and v vector components. Must have same shape and units. If the cubes have geographic coordinates, the values are treated as true distance differentials, e.g. windspeeds, and *not* map coordinate vectors. The components are aligned with the North and East of the cube coordinate system.

Note: At present, if u_cube and v_cube have geographic coordinates, then they must be in a lat-lon coordinate system, though it may be a rotated one. To transform wind values between coordinate systems, use *iris.analysis.cartography.rotate_grid_vectors()*. To transform coordinate grid points, you will need to create 2-dimensional arrays of x and y values. These can be transformed with `cartopy.crs.CRS.transform_points()`.

Kwargs:

- **coords: (list of *Coord* or string)** Coordinates or coordinate names. Use the given coordinates as the axes for the plot. The order of the given coordinates indicates which axis to use for each, where the first element is the horizontal axis of the plot and the second element is the vertical axis of the plot.
- **axes: the `matplotlib.axes.Axes` to use for drawing.** Defaults to the current axes if none provided.

See `matplotlib.pyplot.barbs()` for details of other valid keyword arguments.

---

iris.plot.**citation**(*text*, *figure=None*, *axes=None*)
Add a text citation to a plot.

Places an anchored text citation in the bottom right hand corner of the plot.

Args:

- **text: str** Citation text to be plotted.

Kwargs:

- **figure:matplotlib.figure.Figure** Target figure instance. Defaults to the current figure if none provided.
- **axes: matplotlib.axes.Axes** The axes to use for drawing. Defaults to the current axes if none provided.

---

iris.plot.**contour**(*cube*, **args*, ***kwargs*)
Draws contour lines based on the given Cube.

Kwargs:

- **coords: list of *Coord* objects or coordinate names** Use the given coordinates as the axes for the plot. The order of the given coordinates indicates which axis to use for each, where the first element is the horizontal axis of the plot and the second element is the vertical axis of the plot.
- **axes: matplotlib.axes.Axes** The axes to use for drawing. Defaults to the current axes if none provided.

See matplotlib.pyplot.contour() for details of other valid keyword arguments.

---

iris.plot.**contourf**(*cube*, **args*, ***kwargs*)
Draws filled contours based on the given Cube.

Kwargs:

- **coords: list of *Coord* objects or coordinate names** Use the given coordinates as the axes for the plot. The order of the given coordinates indicates which axis to use for each, where the first element is the horizontal axis of the plot and the second element is the vertical axis of the plot.
- **axes: matplotlib.axes.Axes** The axes to use for drawing. Defaults to the current axes if none provided.

See matplotlib.pyplot.contourf() for details of other valid keyword arguments.

---

iris.plot.**default_projection**(*cube*)
Return the primary map projection for the given cube.

Using the returned projection, one can create a cartopy map with:

```
import matplotlib.pyplot as plt
ax = plt.ax(projection=default_projection(cube))
```

---

iris.plot.**default_projection_extent**(*cube*, *mode=0*)
Return the cube's extents `(x0, x1, y0, y1)` in its default projection.

Keyword arguments:

- **mode: Either `iris.coords.POINT_MODE` or `iris.coords.BOUND_MODE`** Triggers whether the extent should be representative of the cell points, or the limits of the cell's bounds. The default is iris.coords.POINT_MODE.

---

iris.plot.**orography_at_bounds**(*cube*, *facecolor='#888888'*, *coords=None*, *axes=None*)
Plots orography defined at cell boundaries from the given Cube.

---

iris.plot.**orography_at_points**(*cube*, *facecolor='#888888'*, *coords=None*, *axes=None*)
Plots orography defined at sample points from the given Cube.

---

iris.plot.**outline**(*cube*, *coords=None*, *color='k'*, *linewidth=None*, *axes=None*)
Draws cell outlines based on the given Cube.

Kwargs:

- **coords: list of *Coord* objects or coordinate names** Use the given coordinates as the axes for the plot. The order of the given coordinates indicates which axis to use for each, where the first element is the horizontal axis of the plot and the second element is the vertical axis of the plot.
- **color: None or mpl color** The color of the cell outlines. If None, the matplotlibrc setting patch.edgecolor is used by default.
- **linewidth: None or number** The width of the lines showing the cell outlines. If None, the default width in patch.linewidth in matplotlibrc is used.
- **axes: `matplotlib.axes.Axes`** The axes to use for drawing. Defaults to the current axes if none provided.

---

iris.plot.**pcolor**(*cube*, **args*, ***kwargs*)
Draws a pseudocolor plot based on the given 2-dimensional Cube.

The cube must have either two 1-dimensional coordinates or two 2-dimensional coordinates with contiguous bounds to plot the cube against.

Kwargs:

- **coords: list of *Coord* objects or coordinate names** Use the given coordinates as the axes for the plot. The order of the given coordinates indicates which axis to use for each, where the first element is the horizontal axis of the plot and the second element is the vertical axis of the plot.
- **axes: `matplotlib.axes.Axes`** The axes to use for drawing. Defaults to the current axes if none provided.
- **contiguity_tolerance: float** The absolute tolerance used when checking for contiguity between the bounds of the cells. Defaults to None.

See `matplotlib.pyplot.pcolor()` for details of other valid keyword arguments.

iris.plot.**pcolormesh**(*cube*, **args*, ***kwargs*)

Draws a pseudocolor plot based on the given 2-dimensional Cube.

The cube must have either two 1-dimensional coordinates or two 2-dimensional coordinates with contiguous bounds to plot against each other.

Kwargs:

- **coords: list of *Coord* objects or coordinate names** Use the given coordinates as the axes for the plot. The order of the given coordinates indicates which axis to use for each, where the first element is the horizontal axis of the plot and the second element is the vertical axis of the plot.
- **axes: matplotlib.axes.Axes** The axes to use for drawing. Defaults to the current axes if none provided.
- **contiguity_tolerance: float** The absolute tolerance used when checking for contiguity between the bounds of the cells. Defaults to None.

See matplotlib.pyplot.pcolormesh() for details of other valid keyword arguments.

---

iris.plot.**plot**(**args*, ***kwargs*)

Draws a line plot based on the given cube(s) or coordinate(s).

The first one or two arguments may be cubes or coordinates to plot. Each of the following is valid:

```
# plot a 1d cube against its dimension coordinate
plot(cube)

# plot a 1d coordinate
plot(coord)

# plot a 1d cube against a given 1d coordinate, with the cube
# values on the y-axis and the coordinate on the x-axis
plot(coord, cube)

# plot a 1d cube against a given 1d coordinate, with the cube
# values on the x-axis and the coordinate on the y-axis
plot(cube, coord)

# plot two 1d coordinates against one-another
plot(coord1, coord2)

# plot two 1d cubes against one-another
plot(cube1, cube2)
```

Kwargs:

- **axes: matplotlib.axes.Axes** The axes to use for drawing. Defaults to the current axes if none provided.

See matplotlib.pyplot.plot() for details of additional valid keyword arguments.

---

iris.plot.**points**(*cube*, **args*, ***kwargs*)
Draws sample point positions based on the given Cube.

Kwargs:

- **coords: list of *Coord* objects or coordinate names** Use the given coordinates as the axes for the plot. The order of the given coordinates indicates which axis to use for each, where the first element is the horizontal axis of the plot and the second element is the vertical axis of the plot.
- **axes: matplotlib.axes.Axes** The axes to use for drawing. Defaults to the current axes if none provided.

See matplotlib.pyplot.scatter() for details of other valid keyword arguments.

---

iris.plot.**quiver**(*u_cube*, *v_cube*, **args*, ***kwargs*)
Draws an arrow plot from two vector component cubes.

Args:

- **u_cube, v_cube** [*Cube*] u and v vector components. Must have same shape and units. If the cubes have geographic coordinates, the values are treated as true distance differentials, e.g. windspeeds, and *not* map coordinate vectors. The components are aligned with the North and East of the cube coordinate system.

Note: At present, if u_cube and v_cube have geographic coordinates, then they must be in a lat-lon coordinate system, though it may be a rotated one. To transform wind values between coordinate systems, use *iris.analysis.cartography.rotate_grid_vectors()*. To transform coordinate grid points, you will need to create 2-dimensional arrays of x and y values. These can be transformed with cartopy.crs.CRS.transform_points().

Kwargs:

- **coords: list of *Coord* or string** Coordinates or coordinate names. Use the given coordinates as the axes for the plot. The order of the given coordinates indicates which axis to use for each, where the first element is the horizontal axis of the plot and the second element is the vertical axis of the plot.
- **axes: matplotlib.axes.Axes** The axes to use for drawing. Defaults to the current axes if none provided.

See matplotlib.pyplot.quiver() for details of other valid keyword arguments.

---

iris.plot.**scatter**(*x*, *y*, **args*, ***kwargs*)
Draws a scatter plot based on the given cube(s) or coordinate(s).

Args:

- **x: *Cube* or *Coord*** A cube or a coordinate to plot on the x-axis.
- **y: *Cube* or *Coord*** A cube or a coordinate to plot on the y-axis.

Kwargs:

- **axes: matplotlib.axes.Axes** The axes to use for drawing. Defaults to the current axes if none provided.

See matplotlib.pyplot.scatter() for details of additional valid keyword arguments.

---

iris.plot.**symbols**(*x*, *y*, *symbols*, *size*, *axes=None*, *units='inches'*)

Draws fixed-size symbols.

See *iris.symbols* for available symbols.

Args:

- **x: iterable** The x coordinates where the symbols will be plotted.
- **y: iterable** The y coordinates where the symbols will be plotted.
- **symbols: iterable** The symbols (from *iris.symbols*) to plot.
- **size: float** The symbol size in *units*.

Kwargs:

- **axes: matplotlib.axes.Axes** The axes to use for drawing. Defaults to the current axes if none provided.
- **units: ['inches', 'points']** The unit for the symbol size.

PlotDefn(coords, transpose)

class iris.plot.**PlotDefn**(*_cls*, *coords*, *transpose*)

Create new instance of PlotDefn(coords, transpose)

count(*value*, */*)

Return number of occurrences of value.

index(*value*, *start=0*, *stop=9223372036854775807*, */*)

Return first index of value.

Raises ValueError if the value is not present.

coords

Alias for field number 0

transpose

Alias for field number 1

27.17 iris.quickplot

High-level plotting extensions to *iris.plot*.

These routines work much like their *iris.plot* counterparts, but they automatically add a plot title, axis titles, and a colour bar when appropriate.

See also: matplotlib.

In this module:

- *contour*
- *contourf*
- *outline*
- *pcolor*

- *pcolormesh*
- *plot*
- *points*
- *scatter*

iris.quickplot.**contour**(*cube*, **args*, ***kwargs*)

Draws contour lines on a labelled plot based on the given Cube.

With the basic call signature, contour "level" values are chosen automatically:

```
contour(cube)
```

Supply a number to use *N* automatically chosen levels:

```
contour(cube, N)
```

Supply a sequence *V* to use explicitly defined levels:

```
contour(cube, V)
```

See *iris.plot.contour()* for details of valid keyword arguments.

---

iris.quickplot.**contourf**(*cube*, **args*, ***kwargs*)

Draws filled contours on a labelled plot based on the given Cube.

With the basic call signature, contour "level" values are chosen automatically:

```
contour(cube)
```

Supply a number to use *N* automatically chosen levels:

```
contour(cube, N)
```

Supply a sequence *V* to use explicitly defined levels:

```
contour(cube, V)
```

See *iris.plot.contourf()* for details of valid keyword arguments.

---

iris.quickplot.**outline**(*cube*, *coords=None*, *color='k'*, *linewidth=None*, *axes=None*)

Draws cell outlines on a labelled plot based on the given Cube.

Kwargs:

- **coords: list of *Coord* objects or coordinate names** Use the given coordinates as the axes for the plot. The order of the given coordinates indicates which axis to use for each, where the first element is the horizontal axis of the plot and the second element is the vertical axis of the plot.
- **color: None or mpl color** The color of the cell outlines. If None, the matplotlibrc setting patch.edgecolor is used by default.
- **linewidth: None or number** The width of the lines showing the cell outlines. If None, the default width in patch.linewidth in matplotlibrc is used.

---

iris.quickplot.**pcolor**(*cube*, **args*, ***kwargs*)
Draws a labelled pseudocolor plot based on the given Cube.

See *iris.plot.pcolor()* for details of valid keyword arguments.

---

iris.quickplot.**pcolormesh**(*cube*, **args*, ***kwargs*)
Draws a labelled pseudocolour plot based on the given Cube.

See *iris.plot.pcolormesh()* for details of valid keyword arguments.

---

iris.quickplot.**plot**(**args*, ***kwargs*)
Draws a labelled line plot based on the given cube(s) or coordinate(s).

See *iris.plot.plot()* for details of valid arguments and keyword arguments.

---

iris.quickplot.**points**(*cube*, **args*, ***kwargs*)
Draws sample point positions on a labelled plot based on the given Cube.

See *iris.plot.points()* for details of valid keyword arguments.

---

iris.quickplot.**scatter**(*x*, *y*, **args*, ***kwargs*)
Draws a labelled scatter plot based on the given cubes or coordinates.

See *iris.plot.scatter()* for details of valid arguments and keyword arguments.

27.18 iris.std_names

This file contains a dictionary of standard value names that are mapped to another dictionary of other standard name attributes. Currently only the *canonical_unit* exists in these attribute dictionaries.

This file is automatically generated. Do not edit this file by hand.

The file will be generated during a standard build/installation:

```
python setup.py build
python setup.py install
```

Also, the file can be re-generated in the source distribution via:

```
python setup.py std_names
```

Or for more control (e.g. to use an alternative XML file) via:

```
python tools/generate_std_names.py XML_FILE MODULE_FILE
```

In this module:

27.19 iris.symbols

Contains symbol definitions for use with *iris.plot.symbols()*.

In this module:

- *CLOUD_COVER*

iris.symbols.**CLOUD_COVER**
A dictionary mapping WMO cloud cover codes to their corresponding symbol.

See http://www.wmo.int/pages/prog/www/DPFS/documents/485_Vol_I_en_colour.pdf Part II, Appendix II.4, Graphical Representation of Data, Analyses and Forecasts

27.20 iris.time

Time handling.

In this module:

- *PartialDateTime*

A *PartialDateTime* object specifies values for some subset of the calendar/time fields (year, month, hour, etc.) for comparing with `datetime.datetime`-like instances.

Comparisons are defined against any other class with all of the attributes: year, month, day, hour, minute, and second. Notably, this includes `datetime.datetime` and `cftime.datetime`. Comparison also extends to the microsecond attribute for classes, such as `datetime.datetime`, which define it.

A *PartialDateTime* object is not limited to any particular calendar, so no restriction is placed on the range of values allowed in its component fields. Thus, it is perfectly legitimate to create an instance as: *PartialDateTime(month=2, day=30)*.

class iris.time.**PartialDateTime**(*year=None*, *month=None*, *day=None*, *hour=None*, *minute=None*, *second=None*, *microsecond=None*)
Allows partial comparisons against datetime-like objects.

Args:

- year (int):
- month (int):
- day (int):
- hour (int):
- minute (int):
- second (int):
- microsecond (int):

For example, to select any days of the year after the 3rd of April:

```
>>> from iris.time import PartialDateTime
>>> import datetime
>>> pdt = PartialDateTime(month=4, day=3)
>>> datetime.datetime(2014, 4, 1) > pdt
```

(continues on next page)

(continued from previous page)

```
False
>>> datetime.datetime(2014, 4, 5) > pdt
True
>>> datetime.datetime(2014, 5, 1) > pdt
True
>>> datetime.datetime(2015, 2, 1) > pdt
False
```

day
The day number as an integer, or None.

hour
The hour number as an integer, or None.

microsecond
The microsecond number as an integer, or None.

minute
The minute number as an integer, or None.

month
The month number as an integer, or None.

second
The second number as an integer, or None.

timetuple = None

year
The year number as an integer, or None.

27.21 iris.util

Miscellaneous utility functions.

In this module:

- *approx_equal*
- *array_equal*
- *as_compatible_shape*
- *between*
- *broadcast_to_shape*
- *clip_string*
- *column_slices_generator*
- *create_temp_filename*
- *delta*
- *demote_dim_coord_to_aux_coord*
- *describe_diff*
- *equalise_attributes*

- *file_is_newer_than*
- *find_discontiguities*
- *format_array*
- *guess_coord_axis*
- *is_regular*
- *mask_cube*
- *monotonic*
- *new_axis*
- *points_step*
- *promote_aux_coord_to_dim_coord*
- *regular_step*
- *reverse*
- *rolling_window*
- *squeeze*
- *unify_time_units*

iris.util.**approx_equal**(*a*, *b*, *max_absolute_error=1e-10*, *max_relative_error=1e-10*)
Returns whether two numbers are almost equal, allowing for the finite precision of floating point numbers.

---

iris.util.**array_equal**(*array1*, *array2*, *withnans=False*)
Returns whether two arrays have the same shape and elements.

Args:

- **array1, array2 (arraylike):** args to be compared, after normalising with `np.asarray()`.

Kwargs:

- **withnans (bool):** When unset (default), the result is False if either input contains NaN points. This is the normal floating-point arithmetic result. When set, return True if inputs contain the same value in all elements, _including_ any NaN values.

This provides much the same functionality as `numpy.array_equal()`, but with additional support for arrays of strings and NaN-tolerant operation.

---

iris.util.**as_compatible_shape**(*src_cube*, *target_cube*)
Return a cube with added length one dimensions to match the dimensionality and dimension ordering of *target_cube*.

This function can be used to add the dimensions that have been collapsed, aggregated or sliced out, promoting scalar coordinates to length one dimension coordinates where necessary. It operates by matching coordinate metadata to infer the dimensions that need modifying, so the provided cubes must have coordinates with the same metadata (see `iris.common.CoordMetadata`).

Note: This function will load and copy the data payload of *src_cube*.

Deprecated since version 3.0.0: Instead use *Resolve*. For example, rather than calling `as_compatible_shape(src_cube, target_cube)` replace with `Resolve(src_cube, target_cube)(target_cube.core_data())`.

Args:

- **src_cube:** An instance of *iris.cube.Cube* with missing dimensions.
- **target_cube:** An instance of *iris.cube.Cube* with the desired dimensionality.

Returns A instance of *iris.cube.Cube* with the same dimensionality as *target_cube* but with the data and coordinates from *src_cube* suitably reshaped to fit.

`iris.util.`**between**(*lh*, *rh*, *lh_inclusive=True*, *rh_inclusive=True*)

Provides a convenient way of defining a 3 element inequality such as `a < number < b`.

Arguments:

- **lh** The left hand element of the inequality
- **rh** The right hand element of the inequality

Keywords:

- **lh_inclusive - boolean** Affects the left hand comparison operator to use in the inequality. True for <= false for <. Defaults to True.
- **rh_inclusive - boolean** Same as lh_inclusive but for right hand operator.

For example:

```
between_3_and_6 = between(3, 6)
for i in range(10):
   print(i, between_3_and_6(i))


between_3_and_6 = between(3, 6, rh_inclusive=False)
for i in range(10):
   print(i, between_3_and_6(i))
```

`iris.util.`**broadcast_to_shape**(*array*, *shape*, *dim_map*)

Broadcast an array to a given shape.

Each dimension of the array must correspond to a dimension in the given shape. Striding is used to repeat the array until it matches the desired shape, returning repeated views on the original array. If you need to write to the resulting array, make a copy first.

Args:

- **array (numpy.ndarray-like)** An array to broadcast.
- **shape (list, tuple etc.):** The shape the array should be broadcast to.
- **dim_map (list, tuple etc.):** A mapping of the dimensions of *array* to their corresponding element in *shape*. *dim_map* must be the same length as the number of dimensions in *array*. Each element of *dim_map* corresponds to a dimension of *array* and its value provides the index in *shape* which the dimension of *array* corresponds to, so the first element of *dim_map* gives the index of *shape* that corresponds to the first dimension of *array* etc.

Examples:

Broadcasting an array of shape (2, 3) to the shape (5, 2, 6, 3) where the first dimension of the array corresponds to the second element of the desired shape and the second dimension of the array corresponds to the fourth element of the desired shape:

```
a = np.array([[1, 2, 3], [4, 5, 6]])
b = broadcast_to_shape(a, (5, 2, 6, 3), (1, 3))
```

Broadcasting an array of shape (48, 96) to the shape (96, 48, 12):

```
# a is an array of shape (48, 96)
result = broadcast_to_shape(a, (96, 48, 12), (1, 0))
```

---

iris.util.**clip_string**(*the_str*, *clip_length=70*, *rider='...'*)

Returns a clipped version of the string based on the specified clip length and whether or not any graceful clip points can be found.

If the string to be clipped is shorter than the specified clip length, the original string is returned.

If the string is longer than the clip length, a graceful point (a space character) after the clip length is searched for. If a graceful point is found the string is clipped at this point and the rider is added. If no graceful point can be found, then the string is clipped exactly where the user requested and the rider is added.

Args:

- **the_str** The string to be clipped
- **clip_length** The length in characters that the input string should be clipped to. Defaults to a preconfigured value if not specified.
- **rider** A series of characters appended at the end of the returned string to show it has been clipped. Defaults to a preconfigured value if not specified.

Returns The string clipped to the required length with a rider appended. If the clip length was greater than the original string, the original string is returned unaltered.

---

iris.util.**column_slices_generator**(*full_slice*, *ndims*)

Given a full slice full of tuples, return a dictionary mapping old data dimensions to new and a generator which gives the successive slices needed to index correctly (across columns).

This routine deals with the special functionality for tuple based indexing e.g. [0, (3, 5), :, (1, 6, 8)] by first providing a slice which takes the non tuple slices out first i.e. [0, :, :, :] then subsequently iterates through each of the tuples taking out the appropriate slices i.e. [(3, 5), :, :] followed by [:, :, (1, 6, 8)]

This method was developed as numpy does not support the direct approach of [(3, 5), : , (1, 6, 8)] for column based indexing.

---

iris.util.**create_temp_filename**(*suffix=''*)

Return a temporary file name.

Parameters **extension.** (** suffix - Optional filename*) –

iris.util.**delta**(*ndarray*, *dimension*, *circular=False*)

Calculates the difference between values along a given dimension.

Args:

- **ndarray:** The array over which to do the difference.
- **dimension:** The dimension over which to do the difference on ndarray.
- **circular:** If not False then return n results in the requested dimension with the delta between the last and first element included in the result otherwise the result will be of length n-1 (where n is the length of ndarray in the given dimension's direction)

  If circular is numeric then the value of circular will be added to the last element of the given dimension if the last element is negative, otherwise the value of circular will be subtracted from the last element.

  The example below illustrates the process:

```
original array              -180, -90,  0,    90
delta (with circular=360):    90,  90, 90, -270+360
```

Note: The difference algorithm implemented is forward difference:

```
>>> import numpy as np
>>> import iris.util
>>> original = np.array([-180, -90, 0, 90])
>>> iris.util.delta(original, 0)
array([90, 90, 90])
>>> iris.util.delta(original, 0, circular=360)
array([90, 90, 90, 90])
```

iris.util.**demote_dim_coord_to_aux_coord**(*cube*, *name_or_coord*)

Demotes a dimension coordinate on the cube to an auxiliary coordinate.

The DimCoord is demoted to an auxiliary coordinate on the cube. The dimension of the cube that was associated with the DimCoord becomes anonymous. The class of the coordinate is left as DimCoord, it is not recast as an AuxCoord instance.

Args:

- **cube** An instance of *iris.cube.Cube*
- **name_or_coord:** Either

  (a) An instance of *iris.coords.DimCoord*

  or

  (b) the `standard_name`, `long_name`, or `var_name` of an instance of an instance of *iris.coords.DimCoord*.

For example,

```
>>> print(cube)
air_temperature / (K)               (time: 240; latitude: 37; longitude:␣
↪49)
    Dimension coordinates:
        time                            x              -              -
        latitude                        -              x              -
        longitude                       -              -              x
    Auxiliary coordinates:
        forecast_period                 x              -              -
        year                            x              -              -
>>> demote_dim_coord_to_aux_coord(cube, "time")
>>> print(cube)
air_temperature / (K)               (-- : 240; latitude: 37; longitude: 49)
    Dimension coordinates:
        latitude                        -              x              -
        longitude                       -              -              x
    Auxiliary coordinates:
        forecast_period                 x              -              -
        time                            x              -              -
        year                            x              -              -
```

---

iris.util.**describe_diff**(*cube_a*, *cube_b*, *output_file=None*)

Prints the differences that prevent compatibility between two cubes, as defined by *iris.cube.Cube.is_compatible()*.

Args:

- **cube_a:** An instance of *iris.cube.Cube* or `iris.cube.CubeMetadata`.
- **cube_b:** An instance of *iris.cube.Cube* or `iris.cube.CubeMetadata`.
- **output_file:** A `file` or file-like object to receive output. Defaults to sys.stdout.

See also:

iris.cube.Cube.is_compatible()

Note: Compatibility does not guarantee that two cubes can be merged. Instead, this function is designed to provide a verbose description of the differences in metadata between two cubes. Determining whether two cubes will merge requires additional logic that is beyond the scope of this function.

---

iris.util.**equalise_attributes**(*cubes*)

Delete cube attributes that are not identical over all cubes in a group.

This function simply deletes any attributes which are not the same for all the given cubes. The cubes will then have identical attributes. The given cubes are modified in-place.

Args:

- **cubes (iterable of *iris.cube.Cube*):** A collection of cubes to compare and adjust.

---

iris.util.**file_is_newer_than**(*result_path*, *source_paths*)

Return whether the 'result' file has a later modification time than all of the 'source' files.

If a stored result depends entirely on known 'sources', it need only be re-built when one of them changes. This function can be used to test that by comparing file timestamps.

Args:

- **result_path (string):** The filepath of a file containing some derived result data.
- **source_paths (string or iterable of strings):** The path(s) to the original datafiles used to make the result. May include wildcards and '~' expansions (like Iris load paths), but not URIs.

Returns

True if all the sources are older than the result, else False.

If any of the file paths describes no existing files, an exception will be raised.

Note: There are obvious caveats to using file timestamps for this, as correct usage depends on how the sources might change. For example, a file could be replaced by one of the same name, but an older timestamp.

If wildcards and '~' expansions are used, this introduces even more uncertainty, as then you cannot even be sure that the resulting list of file names is the same as the originals. For example, some files may have been deleted or others added.

Note: The result file may often be a `pickle` file. In that case, it also depends on the relevant module sources, so extra caution is required. Ideally, an additional check on iris.__version__ is advised.

---

iris.util.**find_discontiguities**(*cube*, *rel_tol=1e-05*, *abs_tol=1e-08*)

Searches coord for discontiguities in the bounds array, returned as a boolean array (True where discontiguities are present).

Args:

- **cube (*iris.cube.Cube*):** The cube to be checked for discontinuities in its 'x' and 'y' coordinates.

Kwargs:

- **rel_tol (float):** The relative equality tolerance to apply in coordinate bounds checking.
- **abs_tol (float):** The absolute value tolerance to apply in coordinate bounds checking.

Returns:

- **result (*numpy.ndarray* of bool) :** true/false map of which cells in the cube XY grid have discontiguities in the coordinate points array.

  This can be used as the input array for *iris.util.mask_cube()*.

Examples:

```
# Find any unknown discontiguities in your cube's x and y arrays:
discontiguities = iris.util.find_discontiguities(cube)

# Pass the resultant boolean array to `iris.util.mask_cube`
# with a cube slice; this will use the boolean array to mask
# any discontiguous data points before plotting:
masked_cube_slice = iris.util.mask_cube(cube[0], discontiguities)

# Plot the masked cube slice:
iplt.pcolormesh(masked_cube_slice)
```

---

iris.util.**format_array**(*arr*)

Returns the given array as a string, using the python builtin str function on a piecewise basis.

Useful for xml representation of arrays.

For customisations, use the `numpy.core.arrayprint` directly.

---

iris.util.**guess_coord_axis**(*coord*)

Returns a "best guess" axis name of the coordinate.

Heuristic categorisation of the coordinate into either label 'T', 'Z', 'Y', 'X' or None.

Args:

- **coord:** The *iris.coords.Coord*.

Returns 'T', 'Z', 'Y', 'X', or None.

---

iris.util.**is_regular**(*coord*)

Determine if the given coord is regular.

---

iris.util.**mask_cube**(*cube*, *points_to_mask*)

Masks any cells in the data array which correspond to cells marked *True* in the *points_to_mask* array.

Args:

- **cube (*iris.cube.Cube*):** A 2-dimensional instance of *iris.cube.Cube*.
- **points_to_mask (*numpy.ndarray* of bool):** A 2d boolean array of Truth values representing points to mask in the x and y arrays of the cube.

Returns:

- **result (*iris.cube.Cube*):** A cube whose data array is masked at points specified by input array.

---

iris.util.**monotonic**(*array*, *strict=False*, *return_direction=False*)

Return whether the given 1d array is monotonic.

Note that, the array must not contain missing data.

Kwargs:

- **strict (boolean)** Flag to enable strict monotonic checking
- **return_direction (boolean)** Flag to change return behaviour to return (monotonic_status, direction). Direction will be 1 for positive or -1 for negative. The direction is meaningless if the array is not monotonic.

Returns:

- **monotonic_status (boolean)** Whether the array was monotonic.

  If the return_direction flag was given then the returned value will be:

  ```
  (monotonic_status, direction)
  ```

---

iris.util.**new_axis**(*src_cube*, *scalar_coord=None*)

Create a new axis as the leading dimension of the cube, promoting a scalar coordinate if specified.

Args:

- **src_cube (*iris.cube.Cube*)** Source cube on which to generate a new axis.

Kwargs:

- **scalar_coord (iris.coord.Coord or 'string')** Scalar coordinate to promote to a dimension coordinate.

Returns A new *iris.cube.Cube* instance with one extra leading dimension (length 1).

For example:

```
>>> cube.shape
(360, 360)
>>> ncube = iris.util.new_axis(cube, 'time')
>>> ncube.shape
(1, 360, 360)
```

---

iris.util.**points_step**(*points*)

Determine whether a NumPy array has a regular step.

---

iris.util.**promote_aux_coord_to_dim_coord**(*cube*, *name_or_coord*)

Promotes an AuxCoord on the cube to a DimCoord. This AuxCoord must be associated with a single cube dimension. If the AuxCoord is associated with a dimension that already has a DimCoord, that DimCoord gets demoted to an AuxCoord.

Args:

- **cube** An instance of *iris.cube.Cube*
- **name_or_coord:** Either

  (a) An instance of *iris.coords.AuxCoord*

  or

  (b) the `standard_name`, `long_name`, or `var_name` of an instance of an instance of *iris.coords.AuxCoord*.

For example,

```
>>> print(cube)
air_temperature / (K)               (time: 240; latitude: 37; longitude:␣
↪49)
    Dimension coordinates:
        time                            x             -             -
        latitude                        -             x             -
        longitude                       -             -             x
    Auxiliary coordinates:
        forecast_period                 x             -             -
        year                            x             -             -
>>> promote_aux_coord_to_dim_coord(cube, "year")
>>> print(cube)
air_temperature / (K)               (year: 240; latitude: 37; longitude:␣
↪49)
    Dimension coordinates:
        year                            x             -             -
        latitude                        -             x             -
        longitude                       -             -             x
    Auxiliary coordinates:
        forecast_period                 x             -             -
        time                            x             -             -
```

---

iris.util.**regular_step**(*coord*)

Return the regular step from a coord or fail.

---

iris.util.**reverse**(*cube_or_array*, *coords_or_dims*)

Reverse the cube or array along the given dimensions.

Args:

- **cube_or_array: *iris.cube.Cube* or numpy.ndarray** The cube or array to reverse.
- **coords_or_dims: int, str, *iris.coords.Coord* or sequence of these** Identify one or more dimensions to reverse. If cube_or_array is a numpy array, use int or a sequence of ints, as in the examples below. If cube_or_array is a Cube, a Coord or coordinate name (or sequence of these) may be specified instead.

```
>>> import numpy as np
>>> a = np.arange(24).reshape(2, 3, 4)
>>> print(a)
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
>>> print(reverse(a, 1))
[[[ 8  9 10 11]
  [ 4  5  6  7]
```

(continues on next page)

(continued from previous page)

```
  [ 0  1  2  3]]

 [[20 21 22 23]
  [16 17 18 19]
  [12 13 14 15]]]
>>> print(reverse(a, [1, 2]))
[[[11 10  9  8]
  [ 7  6  5  4]
  [ 3  2  1  0]]

 [[23 22 21 20]
  [19 18 17 16]
  [15 14 13 12]]]
```

---

iris.util.**rolling_window**(*a*, *window=1*, *step=1*, *axis=- 1*)

Make an ndarray with a rolling window of the last dimension

Args:

- **a** [array_like] Array to add rolling window to

Kwargs:

- **window** [int] Size of rolling window
- **step** [int] Size of step between rolling windows
- **axis** [int] Axis to take the rolling window over

Returns Array that is a view of the original array with an added dimension of the size of the given window at axis + 1.

Examples:

```
>>> x = np.arange(10).reshape((2, 5))
>>> rolling_window(x, 3)
array([[[0, 1, 2], [1, 2, 3], [2, 3, 4]],
       [[5, 6, 7], [6, 7, 8], [7, 8, 9]]])
```

Calculate rolling mean of last dimension:

```
>>> np.mean(rolling_window(x, 3), -1)
array([[ 1.,  2.,  3.],
       [ 6.,  7.,  8.]])
```

---

iris.util.**squeeze**(*cube*)

Removes any dimension of length one. If it has an associated DimCoord or AuxCoord, this becomes a scalar coord.

Args:

- **cube** (*iris.cube.Cube*) Source cube to remove length 1 dimension(s) from.

Returns A new *iris.cube.Cube* instance without any dimensions of length 1.

For example:

```
>>> cube.shape
(1, 360, 360)
>>> ncube = iris.util.squeeze(cube)
>>> ncube.shape
(360, 360)
```

---

iris.util.**unify_time_units**(*cubes*)

Performs an in-place conversion of the time units of all time coords in the cubes in a given iterable. One common epoch is defined for each calendar found in the cubes to prevent units being defined with inconsistencies between epoch and calendar.

Each epoch is defined from the first suitable time coordinate found in the input cubes.

Arg:

- **cubes:** An iterable containing *iris.cube.Cube* instances.

A package for handling multi-dimensional data and associated metadata.

Note: The Iris documentation has further usage information, including a *user guide* which should be the first port of call for new users.

The functions in this module provide the main way to load and/or save your data.

The *load()* function provides a simple way to explore data from the interactive Python prompt. It will convert the source data into *Cubes*, and combine those cubes into higher-dimensional cubes where possible.

The *load_cube()* and *load_cubes()* functions are similar to *load()*, but they raise an exception if the number of cubes is not what was expected. They are more useful in scripts, where they can provide an early sanity check on incoming data.

The *load_raw()* function is provided for those occasions where the automatic combination of cubes into higher-dimensional cubes is undesirable. However, it is intended as a tool of last resort! If you experience a problem with the automatic combination process then please raise an issue with the Iris developers.

To persist a cube to the file-system, use the *save()* function.

All the load functions share very similar arguments:

- **uris:** Either a single filename/URI expressed as a string, or an iterable of filenames/URIs.

  Filenames can contain ~ or ~*user* abbreviations, and/or Unix shell-style wildcards (e.g. * and *?*). See the standard library function `os.path.expanduser()` and module `fnmatch` for more details.

- **constraints:** Either a single constraint, or an iterable of constraints. Each constraint can be either a string, an instance of *iris.Constraint*, or an instance of *iris.AttributeConstraint*. If the constraint is a string it will be used to match against cube.name().

  For example:

```
# Load air temperature data.
load_cube(uri, 'air_temperature')

# Load data with a specific model level number.
load_cube(uri, iris.Constraint(model_level_number=1))
```

(continues on next page)

(continued from previous page)

```
# Load data with a specific STASH code.
load_cube(uri, iris.AttributeConstraint(STASH='m01s00i004'))
```

- **callback:** A function to add metadata from the originating field and/or URI which obeys the following rules:
  1. Function signature must be: `(cube, field, filename)`.
  2. Modifies the given cube inplace, unless a new cube is returned by the function.
  3. If the cube is to be rejected the callback must raise an *iris.exceptions.IgnoreCubeException*.

  For example:

```
def callback(cube, field, filename):
    # Extract ID from filenames given as: <prefix>__<exp_id>
    experiment_id = filename.split('__')[1]
    experiment_coord = iris.coords.AuxCoord(
        experiment_id, long_name='experiment_id')
    cube.add_aux_coord(experiment_coord)
```

In this module:

- *load*
- *load_cube*
- *load_cubes*
- *load_raw*
- *save*
- *Constraint*
- *AttributeConstraint*
- *NameConstraint*
- *sample_data_path*
- *site_configuration*
- *Future*
- *FUTURE*
- *IrisDeprecation*

iris.**load**(*uris*, *constraints=None*, *callback=None*)

Loads any number of Cubes for each constraint.

For a full description of the arguments, please see the module documentation for *iris*.

Args:

- **uris:** One or more filenames/URIs.

Kwargs:

- **constraints:** One or more constraints.
- **callback:** A modifier/filter function.

Returns An *iris.cube.CubeList*. Note that there is no inherent order to this *iris.cube.CubeList* and it should be treated as if it were random.

iris.**load_cube**(*uris*, *constraint=None*, *callback=None*)
Loads a single cube.

For a full description of the arguments, please see the module documentation for *iris*.

Args:
- **uris:** One or more filenames/URIs.

Kwargs:
- **constraints:** A constraint.
- **callback:** A modifier/filter function.

Returns An *iris.cube.Cube*.

---

iris.**load_cubes**(*uris*, *constraints=None*, *callback=None*)
Loads exactly one Cube for each constraint.

For a full description of the arguments, please see the module documentation for *iris*.

Args:
- **uris:** One or more filenames/URIs.

Kwargs:
- **constraints:** One or more constraints.
- **callback:** A modifier/filter function.

Returns An *iris.cube.CubeList*. Note that there is no inherent order to this *iris.cube.CubeList* and it should be treated as if it were random.

---

iris.**load_raw**(*uris*, *constraints=None*, *callback=None*)
Loads non-merged cubes.

This function is provided for those occasions where the automatic combination of cubes into higher-dimensional cubes is undesirable. However, it is intended as a tool of last resort! If you experience a problem with the automatic combination process then please raise an issue with the Iris developers.

For a full description of the arguments, please see the module documentation for *iris*.

Args:
- **uris:** One or more filenames/URIs.

Kwargs:
- **constraints:** One or more constraints.
- **callback:** A modifier/filter function.

Returns An *iris.cube.CubeList*.

---

iris.**save**(*source*, *target*, *saver=None*, ***kwargs*)
Save one or more Cubes to file (or other writeable).

Iris currently supports three file formats for saving, which it can recognise by filename extension:
- **netCDF - the Unidata network Common Data Format:**
  - see *iris.fileformats.netcdf.save()*
- **GRIB2 - the WMO GRIdded Binary data format:**
  - see `iris_grib.save_grib2()`.
- **PP - the Met Office UM Post Processing Format:**
  - see *iris.fileformats.pp.save()*

A custom saver can be provided to the function to write to a different file format.

Args:

- **source:** *iris.cube.Cube*, *iris.cube.CubeList* or sequence of cubes.
- **target:** A filename (or writeable, depending on file format). When given a filename or file, Iris can determine the file format.

Kwargs:

- **saver:** Optional. Specifies the file format to save. If omitted, Iris will attempt to determine the format.

  If a string, this is the recognised filename extension (where the actual filename may not have it). Otherwise the value is a saver function, of the form: `my_saver(cube, target)` plus any custom keywords. It is assumed that a saver will accept an `append` keyword if it's file format can handle multiple cubes. See also *iris.io.add_saver()*.

All other keywords are passed through to the saver function; see the relevant saver documentation for more information on keyword arguments.

Examples:

```
# Save a cube to PP
iris.save(my_cube, "myfile.pp")

# Save a cube list to a PP file, appending to the contents of the file
# if it already exists
iris.save(my_cube_list, "myfile.pp", append=True)

# Save a cube to netCDF, defaults to NETCDF4 file format
iris.save(my_cube, "myfile.nc")

# Save a cube list to netCDF, using the NETCDF3_CLASSIC storage option
iris.save(my_cube_list, "myfile.nc", netcdf_format="NETCDF3_CLASSIC")
```

Warning: Saving a cube whose data has been loaded lazily (if *cube.has_lazy_data()* returns *True*) to the same file it expects to load data from will cause both the data in-memory and the data on disk to be lost.

```
cube = iris.load_cube("somefile.nc")
# The next line causes data loss in 'somefile.nc' and the cube.
iris.save(cube, "somefile.nc")
```

In general, overwriting a file which is the source for any lazily loaded data can result in corruption. Users should proceed with caution when attempting to overwrite an existing file.

Constraints are the mechanism by which cubes can be pattern matched and filtered according to specific criteria.

Once a constraint has been defined, it can be applied to cubes using the *Constraint.extract()* method.

class iris.**Constraint**(*name=None*, *cube_func=None*, *coord_values=None*, ***kwargs*)

Creates a new instance of a Constraint which can be used for filtering cube loading or cube list extraction.

Args:

- **name: string or None** If a string, it is used as the name to match against the *~iris.cube.Cube.names* property.
- **cube_func: callable or None** If a callable, it must accept a Cube as its first and only argument and return either True or False.

- **coord_values: dict or None** If a dict, it must map coordinate name to the condition on the associated coordinate.
- ****kwargs*:** The remaining keyword arguments are converted to coordinate constraints. The name of the argument gives the name of a coordinate, and the value of the argument is the condition to meet on that coordinate:

```
Constraint(model_level_number=10)
```

Coordinate level constraints can be of several types:

- **string, int or float** - the value of the coordinate to match. e.g. `model_level_number=10`
- **list of values** - the possible values that the coordinate may have to match. e.g. `model_level_number=[10, 12]`
- **callable** - a function which accepts a *iris.coords.Cell* instance as its first and only argument returning True or False if the value of the Cell is desired. e.g. `model_level_number=lambda cell: 5 < cell < 10`

The *user guide* covers cube much of constraining in detail, however an example which uses all of the features of this class is given here for completeness:

```
Constraint(name='air_potential_temperature',
           cube_func=lambda cube: cube.units == 'kelvin',
           coord_values={'latitude':lambda cell: 0 < cell < 90},
           model_level_number=[10, 12])
           & Constraint(ensemble_member=2)
```

Constraint filtering is performed at the cell level. For further details on how cell comparisons are performed see *iris.coords.Cell*.

extract(*cube*)
Return the subset of the given cube which matches this constraint, else return None.

Provides a simple Cube-attribute based *Constraint*.

class iris.**AttributeConstraint**(***attributes*)
Example usage:

```
iris.AttributeConstraint(STASH='m01s16i004')

iris.AttributeConstraint(
    STASH=lambda stash: str(stash).endswith('i005'))
```

Note: Attribute constraint names are case sensitive.

extract(*cube*)
Return the subset of the given cube which matches this constraint, else return None.

Provides a simple Cube name based *Constraint*.

class iris.**NameConstraint**(*standard_name='none'*, *long_name='none'*, *var_name='none'*, *STASH='none'*)

Provides a simple Cube name based *Constraint*, which matches against each of the names provided, which may be either standard name, long name, NetCDF variable name and/or the STASH from the attributes dictionary.

The name constraint will only succeed if *all* of the provided names match.

Kwargs:

- **standard_name:** A string or callable representing the standard name to match against.
- **long_name:** A string or callable representing the long name to match against.
- **var_name:** A string or callable representing the NetCDF variable name to match against.
- **STASH:** A string or callable representing the UM STASH code to match against.

Note: The default value of each of the keyword arguments is the string "none", rather than the singleton None, as None may be a legitimate value to be matched against e.g., to constrain against all cubes where the standard_name is not set, then use standard_name=None.

Returns:

- Boolean

Example usage:

```
iris.NameConstraint(long_name='air temp', var_name=None)

iris.NameConstraint(long_name=lambda name: 'temp' in name)

iris.NameConstraint(standard_name='air_temperature',
                    STASH=lambda stash: stash.item == 203)
```

extract(*cube*)

Return the subset of the given cube which matches this constraint, else return None.

---

iris.**sample_data_path**(**path_to_join*)

Given the sample data resource, returns the full path to the file.

Note: This function is only for locating files in the iris sample data collection (installed separately from iris). It is not needed or appropriate for general file access.

---

iris.**site_configuration**

Iris site configuration dictionary.

Run-time configuration controller.

class iris.**Future**

A container for run-time options controls.

To adjust the values simply update the relevant attribute from within your code. For example:

```
iris.FUTURE.example_future_flag = False
```

If Iris code is executed with multiple threads, note the values of these options are thread-specific.

Note: iris.FUTURE.example_future_flag does not exist. It is provided as an example because there are currently no flags in iris.Future.

---

context(***kwargs*)

Return a context manager which allows temporary modification of the option values for the active thread.

On entry to the *with* statement, all keyword arguments are applied to the Future object. On exit from the *with* statement, the previous state is restored.

For example::

with iris.FUTURE.context(example_future_flag=False): # ... code that expects some past behaviour

---

Note: iris.FUTURE.example_future_flag does not exist and is provided only as an example since there are currently no flags in Future.

---

deprecated_options = {}

---

iris.**FUTURE**

Object containing all the Iris run-time options.

An Iris deprecation warning.

class iris.**IrisDeprecation**

An Iris deprecation warning.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

args

CHAPTER

TWENTYEIGHT

WHAT'S NEW IN IRIS

These "What's new" pages describe the important changes between major Iris versions.

28.1 v3.1 (17 Sep 2021)

This document explains the changes made to Iris for this release (*View all changes*.)

v3.1.0 Release Highlights

The highlights for this minor release of Iris include:

- Much faster import times, from minimised loading of submodules. However, note that this may break existing code which does not declare all imports. See *minimised imports*.
- Speedup for loading single phenomena from netcdf. See *netcdf load speedup*.
- We've dropped support for Python 3.6. See *minimum Python version 3.7*.
- Updated formatting of cube printouts. See *new-style cube printouts*.
- Multiple improvements to developer guide documentation. See entries in the *"Documentation" section*, below.

And finally, get in touch with us on GitHub if you have any issues or feature requests for improving Iris. Enjoy!

28.1.1 Announcements

1. Congratulations to @jamesp who recently became an Iris core developer after joining the Iris development team at the Met Office.
2. A special thanks goes to @akuhnregnier, @gcaria, @jamesp, @schlunma, @MHBalsmeier and @Badboy-16 all of whom made their first contributions to Iris, which were gratefully received and included in this release. Keep up the awesome work!

28.1.2 Features

1. @pelson and @trexfeathers enhanced *`iris.plot.plot()`* and *`iris.quickplot.plot()`* to automatically place the cube on the x axis if the primary coordinate being plotted against is a vertical coordinate. E.g. `iris.plot.plot(z_cube)` will produce a z-vs-phenomenon plot, where before it would have produced a phenomenon-vs-z plot. (PR #3906)
2. @jonseddon added *`iris.plot.barbs()`* to provide a convenient way to use `matplotlib.pyplot.barbs()` with Iris cubes. A gallery example was included to illustrate the new method's use. (PR #3710)

3. @bjlittle introduced *iris.common.metadata.hexdigest()* to the public API. Previously it was a private function introduced in `v3.0.0`. Given any object, *hexdigest()* returns a string representation of the 64-bit non-cryptographic hash of the object using the extremely fast xxhash hashing algorithm. (PR #4020)
4. @rcomer implemented a `__str__` method for metadata classes, so printing these objects skips metadata elements that are set to None or an empty string or dictionary. (PR #4040)
5. @Badboy-16 implemented a `CubeList.copy()` method to return a `CubeList` object instead of a `list`. (PR #4094)
6. @pp-mo and @trexfeathers reformatted *iris.cube.Cube.summary()*, (which is used for `print(Cube)`); putting *cell_methods* before *attributes*, and improving spacing throughout. (PR #4206)
7. @schlunma added support for loading atmosphere sigma coordinates from netcdf-CF files. These now load as *iris.aux_factory.AtmosphereSigmaFactory* derived coordinates. (PR #4052)

28.1.3 Bugs Fixed

1. @gcaria fixed *Cell* comparison with 0-dimensional arrays and 1-dimensional arrays with len=1. (PR #4083)
2. @gcaria fixed *cell_measure_dims()* to also accept the string name of a *CellMeasure*. (PR #3931)
3. @gcaria fixed *ancillary_variable_dims()* to also accept the string name of a *AncillaryVariable*. (PR #3931)
4. @rcomer modified *contourf()* to skip the special handling for antialiasing when data values are too low for it to have an effect. This caused unexpected artifacts in some edge cases, as shown at Issue #4086. (PR #4150)
5. @MHBalsmeier modified *contourf()* to generalize PR #4150 for the cases where NaN values occur in the plot array (PR #4263)
6. @trexfeathers fixed the "anomaly_log_colouring" gallery example to be compatible with the latest Matplotlib usage (PR #4115)

28.1.4 Performance Enhancements

1. @bjlittle added support for automated `import` linting with isort, which also includes significant speed-ups for Iris imports. (PR #4174)
2. @bjlittle Optimised the creation of dynamic metadata manager classes within the *metadata_manager_factory()*, resulting in a significant speed-up in the creation of Iris *AncillaryVariable*, *AuxCoord*, *CellMeasure*, and *Cube* instances. (PR #4227)
3. @pp-mo and @lbdreyer optimised loading netcdf files, resulting in a speed up when loading with a single *NameConstraint()*.

   For example, `cube = iris.load('x.nc', NameConstraint('air_temperature'))`.

   Note that this optimisation only applies when matching on standard name, long name or NetCDF variable name, not when matching on STASH. (PR #4176)

28.1.5 Incompatible Changes

1. The *adoption of 'isort'* has significantly reduced the import time of Iris packages.

   However, this may break existing code which, for convenience, relies on some subpackages being imported implicitly (as some, but not all, previously were).

   For example: `import iris; print(iris.cube.Cube)`.

   This style is essentially unsafe, and in this case no longer works. It must be modified to explicitly import all subpackages,

   i.e. `import iris.cube; print(iris.cube.Cube)`.

28.1.6 Dependencies

1. @bjlittle dropped both black and flake8 package dependencies from our conda YAML and `setup.cfg` PyPI requirements. (PR #4181)
2. @pp-mo removed dependency on PyKE. (PR #4198)

28.1.7 Documentation

1. @rcomer updated the "Seasonal ensemble model plots" and "Global average annual temperature maps" Gallery examples. (PR #3933 and PR #3934)
2. @MHBalsmeier described non-conda installation on Debian-based distros. (PR #3958)
3. @bjlittle clarified in the doc-string that *Coord* is now an abstract base class since Iris `3.0.0`, and it is **not** possible to create an instance of it. (PR #3971)
4. @bjlittle added automated Iris version discovery for the `latest.rst` in the `whatsnew` documentation. (PR #3981)
5. @tkknight stated the Python version used to build the documentation on *Installing Iris* and to the footer of all pages. Also added the copyright years to the footer. (PR #3989)
6. @bjlittle updated the `intersphinx_mapping` and fixed documentation to use `stable` URLs for matplotlib. (PR #4003)
7. @bjlittle added the badge to the README.md. (PR #4004)
8. @tkknight added a banner at the top of every page of the unreleased development documentation if being viewed on Read the Docs. (PR #3999)
9. @bjlittle added post-release instructions on how to *Update PyPI* with scitools-iris. (PR #4038)
10. @bjlittle added the badge to the README.md. See *pre-commit CI* for further details. (PR #4061)
11. @rcomer tweaked docstring layouts in the *iris.plot* module, so they render better in the published documentation. See Issue #4085. (PR #4100)
12. @tkknight documented the `--force` command line option when creating a conda development environment. See *Installing from Source with Conda (Developers)*. (PR #4240)
13. @MHBalsmeier updated and simplified non-conda installation on Debian-based distros. (PR #4260)
14. @bjlittle updated the `intersphinx_mapping` and fixed documentation to use `stable` URLs for matplotlib. (PR #4003)

28.1.8 Internal

1. @rcomer removed an old unused test file. (PR #3913)
2. @tkknight moved the `docs/iris` directory to be in the parent directory `docs`. (PR #3975)
3. @jamesp updated a test for numpy `1.20.0`. (PR #3977)
4. @bjlittle and @jamesp extended the cirrus-ci testing and nox testing automation to support Python 3.8. (PR #3976)
5. @bjlittle rationalised the `noxfile.py`, and added the ability for each `nox` session to list its `conda` environment packages and environment info. (PR #3990)
6. @bjlittle enabled cirrus-ci compute credits for non-draft pull-requests from collaborators targeting the Iris `main` branch. (PR #4007)
7. @akuhnregnier replaced deprecated numpy 1.20 aliases for builtin types. (PR #3997)
8. @bjlittle added conditional task execution to .cirrus.yml to allow developers to easily disable cirrus-ci tasks. See *Skipping Cirrus-CI Tasks*. (PR #4019)
9. @bjlittle and @jamesp addressed a regression in behaviour when using conda 4.10.0 within cirrus-ci. (PR #4084)
10. @bjlittle updated the perceptual imagehash graphical test support for matplotlib 3.4.1. (PR #4087)
11. @jamesp switched cirrus-ci testing and nox testing to use conda-lock files for static test environments. (PR #4108)
12. @bjlittle updated the `bug-report` and `feature-request` GitHub issue templates to remove an external URL reference that caused un-posted user issue content to be lost in the browser when followed. (PR #4147)
13. @bjlittle dropped Python 3.6 support, and automated the discovery of supported Python versions tested by cirrus-ci for documentation. (PR #4163)
14. @bjlittle refactored `setup.py` into `setup.cfg`. (PR #4168)
15. @bjlittle consolidated the `.flake8` configuration into `setup.cfg`. (PR #4200)
16. @bjlittle renamed `iris/master` branch to `iris/main` and migrated references of `master` to `main` within codebase. (PR #4202)
17. @bjlittle added the blacken-docs `pre-commit` hook to automate `black` linting of documentation code blocks. (PR #4205)
18. @bjlittle consolidated nox `black`, `flake8` and `isort` sessions into one `lint` session using `pre-commit`. (PR #4181)
19. @bjlittle streamlined the cirrus-ci testing by removing the `minimal` tests, which are a subset of the `full` tests. (PR #4218)
20. @bjlittle consolidated the cirrus-ci documentation `doctest` and `gallery` tasks into a single task and associated nox session. (PR #4219)
21. @jamesp and @trexfeathers implemented a benchmarking CI check using asv. (PR #4253)
22. @pp-mo and @stephenworsley refactored almost all of *`iris.cube.Cube.summary()`* into the new private module: `iris._representation`; rewritten with a more modular approach, resulting in more readable and extensible code. (PR #3987) (PR #4206)
23. @pp-mo reworked the netcdf loading code, replacing Pyke rules with a pure Python implementation. (PR #4198)
24. @lbdreyer updated the CF standard name table to the latest version: v77. (PR #4282)
25. @jamesp updated a test to the latest numpy version (PR #3977)

26. @bjlittle enabled cirrus-ci compute credits for non-draft pull-requests from collaborators targeting the Iris `master` branch. (PR #4007)

27. @bjlittle added conditional task execution to .cirrus.yml to allow developers to easily disable cirrus-ci tasks. See *Skipping Cirrus-CI Tasks*. (PR #4019)

28.2 v3.0 (25 Jan 2021)

This document explains the changes made to Iris for this release (*View all changes*.)

v3.0.0 Release Highlights

The highlights for this major release of Iris include:

- We've finally dropped support for `Python 2`, so welcome to `Iris 3` and `Python 3`!
- We've extended our coverage of the CF Conventions and Metadata by introducing support for CF Ancillary Data and Quality Flags,
- Lazy regridding is now available for several regridding schemes,
- Managing and manipulating metadata within Iris is now easier and more consistent thanks to the introduction of a new common metadata API,
- *Cube arithmetic* has been significantly improved with regards to extended broadcasting, auto-transposition and a more lenient behaviour towards handling metadata and coordinates,
- Our *documentation* has been refreshed, restructured, revitalised and rehosted on readthedocs,
- It's now easier than ever to *install Iris* as a user or a developer, and the newly revamped developers guide walks you though how you can *get involved* and contribute to Iris,
- Also, this is a major release of Iris, so please be aware of the *incompatible changes* and *deprecations*.

And finally, get in touch with us on GitHub if you have any issues or feature requests for improving Iris. Enjoy!

28.2.1 v3.0.1 (27 Jan 2021)

v3.0.1 Patches

The patches included in this release include:

Internal

1. @bjlittle gracefully promote formula terms within *aux_factory* that have `units` of `unknown` to `units` of `1` (dimensionless), where the formula term **must** have dimensionless `units`. Without this graceful treatment of `units` the resulting *Cube* will **not** contain the expected auxiliary factory, and the associated derived coordinate will be missing. (PR #3965)

28.2.2 v3.0.2 (27 May 2021)

v3.0.2 Patches

The patches included in this release include:

Bugs Fixed

1. @jonseddon handled a malformed `um_stash_source` CF variable attribute in a netCDF file rather than raising a `ValueError`. (PR #4035)
2. @rcomer fixed *`intersection()`* for special cases where one cell's bounds align with the requested maximum and minimum, as reported in Issue #3391. (PR #4059)
3. @bjlittle resolved a regression in arithmetic behaviour between a coordinate and a cube which resulted in a `NotYetImplementedError` being raised, as reported in Issue #4000. This fix supports +, -, *, and / operations between a coordinate and a cube, and for convenience additionally includes `iris.cube.Cube.__neg__()` support. (PR #4159)

Documentation

1. @bjlittle updated the `intersphinx_mapping` and fixed documentation to use `stable` URLs for matplotlib. (PR #4003) [`pre-v3.1.0`]

Internal

1. @jamesp updated a test to the latest numpy version (PR #3977) [`pre-v3.1.0`]
2. @bjlittle enabled cirrus-ci compute credits for non-draft pull-requests from collaborators targeting the Iris `master` branch. (PR #4007) [`pre-v3.1.0`]
3. @bjlittle added conditional task execution to .cirrus.yml to allow developers to easily disable cirrus-ci tasks. See *Skipping Cirrus-CI Tasks*. (PR #4019) [`pre-v3.1.0`]
4. @pp-mo adjusted the use of `dask.array.from_array()` in `iris._lazy_data.as_lazy_data()`, to avoid the dask 'test access'. This makes loading of netcdf files with a large number of variables significantly faster. (PR #4135)
5. @pp-mo reverted a change made previously in (PR #3659) to `iris.fileformats.pp.PPDataProxy.__getitem__()`. The check for empty slicings is no longer needed since (PR #4135) was added. (PR #4141)

Note that, the above contributions labelled with `pre-v3.1.0` are part of the forthcoming Iris v3.1.0 release, but require to be included in this patch release.

28.2.3 v3.0.3 (07 July 2021)

v3.0.3 Patches

The patches included in this release include:

Bugs Fixed

1. @lbdreyer modified *`intersection()`* to use a tolerant equality check, when looking for cells that straddle the wrapping point. (PR #4220)

28.2.4 v3.0.4 (22 July 2021)

v3.0.4 Patches

The patches included in this release include:

Bugs Fixed

1. @pp-mo fixed 2 bugs in cube printout: Firstly, ancillary-variables or cell-measures with long names can now widen the cube "dimensions map" to fit, whereas previously printing these cases caused an Exception. Secondly, cube units are now always printed, whereas previously they were missed out any time that the "dimensions map" was widened to accomodate long coordinate names. (PR #4233)(PR #4238)

Internal

1. @bjlittle Unpinned the cftime package dependency within Iris in order to allow use of the latest versions of cftime, cf-units and nc-time-axis. (PR #4222)
2. @rcomer modified test modules so they run consistently under `pytest` and `nose`, and also fixed some minor issues with *`PartialDateTime`*. (PR #4249)

Note that, we are forced to drop support for `Python 3.6` in this patch due to the third-party package dependencies required by (PR #4222).

28.2.5 Announcements

1. Congratulations to @bouweandela, @jvegasbsc, and @zklaus who recently became Iris core developers. They bring a wealth of expertise to the team, and are using Iris to underpin ESMValTool - "*A community diagnostic and performance metrics tool for routine evaluation of Earth system models in CMIP*". Welcome aboard!
2. Congratulations also goes to @jonseddon who recently became an Iris core developer. We look forward to seeing more of your awesome contributions!

28.2.6 Features

1. @MoseleyS greatly enhanced the *`nimrod`* module to provide richer meta-data translation when loading `Nimrod` data into cubes. This covers most known operational use-cases. (PR #3647)
2. @stephenworsley improved the handling of *`iris.coords.CellMeasure`*s in the *`Cube`* statistical operations *`collapsed()`*, *`aggregated_by()`* and *`rolling_window()`*. These previously removed every *`CellMeasure`* attached to the cube. Now, a *`CellMeasure`* will only be removed if it is associated with an axis over which the statistic is being run. (PR #3549)
3. @stephenworsley, @pp-mo and @abooton added support for CF Ancillary Data variables. These are created as *`iris.coords.AncillaryVariable`*, and appear as components of cubes much like *`AuxCoord`*s, with the new *`Cube`* methods *`add_ancillary_variable()`*, *`remove_ancillary_variable()`*, *`ancillary_variable()`*, *`ancillary_variables()`* and *`ancillary_variable_dims()`*. They are loaded from and saved to NetCDF-CF files. Special support for Quality Flags is also provided, to ensure they load and save with appropriate units. (PR #3800)
4. @bouweandela implemented lazy regridding for the *`Linear`*, *`Nearest`*, and *`AreaWeighted`* regridding schemes. (PR #3701)
5. @bjlittle added logging support within *`iris.analysis.maths`*, *`iris.common.metadata`*, and *`iris.common.resolve`*. Each module defines a `logging.Logger` instance called `logger` with a default `level` of `INFO`. To enable `DEBUG` logging use `logger.setLevel("DEBUG")`. (PR #3785)

6. @bjlittle added the *iris.common.resolve* module, which provides infrastructure to support the analysis, identification and combination of metadata common between two *Cube* operands into a single resultant *Cube* that will be auto-transposed, and with the appropriate broadcast shape. (PR #3785)

7. @bjlittle added the *common metadata API*, which provides a unified treatment of metadata across Iris, and allows users to easily manage and manipulate their metadata in a consistent way. (PR #3785)

8. @bjlittle added *lenient metadata* support, to allow users to control **strict** or **lenient** metadata equivalence, difference and combination. (PR #3785)

9. @bjlittle added *lenient cube maths* support and resolved several long standing major issues with cube arithmetic regarding a more robust treatment of cube broadcasting, cube dimension auto-transposition, and preservation of common metadata and coordinates during cube math operations. Resolves Issue #1887, Issue #2765, and Issue #3478. (PR #3785)

10. @pp-mo and @TomekTrzeciak enhanced `collapse()` to allow a 1-D weights array when collapsing over a single dimension. Previously, the weights had to be the same shape as the whole cube, which could cost a lot of memory in some cases. The 1-D form is supported by most weighted array statistics (such as `np.average()`), so this now works with the corresponding Iris schemes (in that case, *MEAN*). (PR #3943)

28.2.7 Bugs Fixed

1. @stephenworsley fixed *remove_coord()* to now also remove derived coordinates by removing aux_factories. (PR #3641)

2. @jonseddon fixed `isinstance(cube, collections.Iterable)` to now behave as expected if a *Cube* is iterated over, while also ensuring that `TypeError` is still raised. (Fixed by setting the `__iter__()` method in *Cube* to `None`). (PR #3656)

3. @stephenworsley enabled cube concatenation along an axis shared by cell measures; these cell measures are now concatenated together in the resulting cube. Such a scenario would previously cause concatenation to inappropriately fail. (PR #3566)

4. @stephenworsley newly included *CellMeasure*s in *Cube* copy operations. Previously copying a *Cube* would ignore any attached *CellMeasure*. (PR #3546)

5. @bjlittle set a *CellMeasure*'s `measure` attribute to have a default value of `area`. Previously, the `measure` was provided as a keyword argument to *CellMeasure* with a default value of `None`, which caused a `TypeError` when no `measure` was provided, since `area` or `volume` are the only accepted values. (PR #3533)

6. @trexfeathers set **all** plot types in *iris.plot* to now use `matplotlib.dates.date2num` to format date/time coordinates for use on a plot axis (previously *pcolor()* and *pcolormesh()* did not include this behaviour). (PR #3762)

7. @trexfeathers changed date/time axis labels in *iris.quickplot* to now **always** be based on the `epoch` used in `matplotlib.dates.date2num` (previously would take the unit from a time coordinate, if present, even though the coordinate's value had been changed via `date2num`). (PR #3762)

8. @pp-mo newly included attributes of cell measures in NETCDF-CF file loading; they were previously being discarded. They are now available on the *CellMeasure* in the loaded *Cube*. (PR #3800)

9. @pp-mo fixed the netcdf loader to now handle any grid-mapping variables with missing `false_easting` and `false_northing` properties, which was previously failing for some coordinate systems. See Issue #3629. (PR #3804)

10. @stephenworsley changed the way tick labels are assigned from string coords. Previously, the first tick label would occasionally be duplicated. This also removes the use of the deprecated matplotlib `IndexFormatter`. (PR #3857)

11. @znicholls fixed `_title()` to only check `units.is_time_reference` if the `units` symbol is not used. (PR #3902)

12. @rcomer fixed a bug whereby numpy array type attributes on a cube's coordinates could prevent printing it. See Issue #3921. (PR #3922)

28.2.8 Incompatible Changes

1. @pp-mo rationalised *CubeList* extraction methods:

   The former method `iris.cube.CubeList.extract_strict`, and the `strict` keyword of the *extract()* method have been removed, and are replaced by the new routines *extract_cube()* and *extract_cubes()*. The new routines perform the same operation, but in a style more like other `Iris` functions such as *load_cube()* and *load_cubes()*. Unlike `strict` extraction, the type of return value is now completely consistent : *extract_cube()* always returns a *Cube*, and *extract_cubes()* always returns an *iris.cube.CubeList* of a length equal to the number of constraints. (PR #3715)

2. @pp-mo removed the former function `iris.analysis.coord_comparison`. (PR #3562)

3. @bjlittle moved the *iris.experimental.equalise_cubes.equalise_attributes()* function from the *iris.experimental* module into the *iris.util* module. Please use the *iris.util.equalise_attributes()* function instead. (PR #3527)

4. @bjlittle removed the module `iris.experimental.concatenate`. In `v1.6.0` the experimental `concatenate` functionality was moved to the *iris.cube.CubeList.concatenate()* method. Since then, calling the `iris.experimental.concatenate.concatenate()` function raised an exception. (PR #3523)

5. @stephenworsley changed the default units of *DimCoord* and *AuxCoord* from *"1"* to *"unknown"*. (PR #3795)

6. @stephenworsley changed Iris objects loaded from NetCDF-CF files to have `units='unknown'` where the corresponding NetCDF variable has no `units` property. Previously these cases defaulted to `units='1'`. This affects loading of coordinates whose file variable has no "units" attribute (not valid, under CF units rules): These will now have units of *"unknown"*, rather than *"1"*, which **may prevent the creation of a hybrid vertical coordinate**. While these cases used to "work", this was never really correct behaviour. (PR #3795)

7. @SimonPeatman added attribute `var_name` to coordinates created by the *iris.analysis.trajectory.interpolate()* function. This prevents duplicate coordinate errors in certain circumstances. (PR #3718)

8. @bjlittle aligned the *iris.analysis.maths.apply_ufunc()* with the rest of the *iris.analysis.maths* API by changing its keyword argument from `other_cube` to `other`. (PR #3785)

9. @bjlittle changed the *iris.analysis.maths.IFunc.__call__()* to ignore any surplus `other` keyword argument for a `data_func` that requires **only one** argument. This aligns the behaviour of *iris.analysis.maths.IFunc.__call__()* with *apply_ufunc()*. Previously a `ValueError` exception was raised. (PR #3785)

28.2.9 Deprecations

1. @stephenworsley removed the deprecated *iris.Future* flags `cell_date_time_objects`, `netcdf_promote`, `netcdf_no_unlimited` and `clip_latitudes`. (PR #3459)

2. @stephenworsley changed *iris.fileformats.pp.PPField.lbproc* to be an `int`. The deprecated attributes `flag1`, `flag2` etc. have been removed from it. (PR #3461)

3. @bjlittle deprecated *as_compatible_shape()* in preference for *Resolve* e.g., `Resolve(src, tgt)(tgt.core_data())`. The *as_compatible_shape()* function will be removed in a future release of Iris. (PR #3892)

28.2.10 Dependencies

1. @stephenworsley, @trexfeathers and @bjlittle removed `Python2` support, modernising the codebase by switching to exclusive `Python3` support. (PR #3513)
2. @bjlittle improved the developer set up process. Configuring Iris and *Installing from Source with Conda (Developers)* as a developer with all the required package dependencies is now easier with our curated conda environment YAML files. (PR #3812)
3. @stephenworsley pinned Iris to require Dask `>=2.0`. (PR #3460)
4. @stephenworsley and @trexfeathers pinned Iris to require Cartopy `>=0.18`, in order to remain compatible with the latest version of matplotlib. (PR #3762)
5. @bjlittle unpinned Iris to use the latest version of matplotlib. Supporting `Iris` for both `Python2` and `Python3` had resulted in pinning our dependency on matplotlib at `v2.x`. But this is no longer necessary now that `Python2` support has been dropped. (PR #3468)
6. @stephenworsley and @trexfeathers unpinned Iris to use the latest version of Proj. (PR #3762)
7. @stephenworsley and @trexfeathers removed GDAL from the extensions dependency group. We no longer consider it to be an extension. (PR #3762)

28.2.11 Documentation

1. @tkknight moved the *Tri-Polar Grid Projected Plotting* from the general part of the gallery to oceanography. (PR #3761)
2. @tkknight updated documentation to use a modern sphinx theme and be served from https://scitools-iris.readthedocs.io/en/latest/. (PR #3752)
3. @bjlittle added support for the black code formatter. This is now automatically checked on GitHub PRs, replacing the older, unittest-based `iris.tests.test_coding_standards.TestCodeFormat`. Black provides automatic code format correction for most IDEs. See the new developer guide section on *Code Formatting*. (PR #3518)
4. @tkknight and @trexfeathers refreshed the *Contributing a "What's New" Entry* for the *What's New in Iris*. This includes always creating the `latest` what's new page so it appears on the latest documentation at https://scitools-iris.readthedocs.io/en/latest/whatsnew. This resolves Issue #2104, Issue #3451, Issue #3818, Issue #3837. Also updated the *Maintainer Steps* to follow when making a release. (PR #3769, PR #3838, PR #3843)
5. @tkknight enabled the PDF creation of the documentation on the Read the Docs service. The PDF may be accessed by clicking on the version at the bottom of the side bar, then selecting `PDF` from the `Downloads` section. (PR #3765)
6. @stephenworsley added a warning to the *`iris.analysis.cartography.project()`* function regarding its behaviour on projections with non-rectangular boundaries. (PR #3762)
7. @stephenworsley added the *Combining Units* section to the user guide to clarify how `Units` are handled during cube arithmetic. (PR #3803)
8. @tkknight overhauled the *Further Topics* including information on getting involved in becoming a contributor and general structure of the guide. This resolves Issue #2170, Issue #2331, Issue #3453, Issue #314, Issue #2902. (PR #3852)
9. @rcomer added argument descriptions to the *`DimCoord`* docstring. (PR #3681)
10. @tkknight added two url's to be ignored for the `make linkcheck`. This will ensure the Iris github project is not repeatedly hit during the linkcheck for issues and pull requests as it can result in connection refused and thus travis-ci job failures. For more information on linkcheck, see *Testing*. (PR #3873)

11. @tkknight enabled the napolean package that is used by sphinx to cater for the existing google style docstrings and to also allow for numpy docstrings. This resolves Issue #3841. (PR #3871)
12. @tkknight configured `sphinx-build` to promote warnings to errors when building the documentation via `make html`. This will minimise technical debt accruing for the documentation. (PR #3877)
13. @tkknight updated *Installing Iris* to include a reference to Windows Subsystem for Linux. (PR #3885)
14. @tkknight updated the *Iris 3.1.0* homepage to include panels so the links are more visible to users. This uses the sphinx-panels extension. (PR #3884)
15. @bjlittle created the *Further topics* section and included documentation for *Metadata*, *Lenient Metadata*, and *Lenient Cube Maths*. (PR #3890)
16. @jonseddon updated the CF version of the netCDF saver in the *Saving Iris Cubes* section and in the equivalent function docstring. (PR #3925)
17. @bjlittle applied Title Case Capitalization to the documentation. (PR #3940)

28.2.12 Internal

1. @pp-mo and @lbdreyer removed all Iris test dependencies on iris-grib by transferring all relevant content to the iris-grib repository. (PR #3662, PR #3663, PR #3664, PR #3665, PR #3666, PR #3669, PR #3670, PR #3671, PR #3672, PR #3742, PR #3746)
2. @lbdreyer and @pp-mo overhauled the handling of dimensional metadata to remove duplication. (PR #3422, PR #3551)
3. @trexfeathers simplified the standard license header for all files, which removes the need to repeatedly update year numbers in the header. (PR #3489)
4. @stephenworsley changed the numerical values in tests involving the Robinson projection due to improvements made in Proj. (PR #3762) (see also Proj#1292 and Proj#2151)
5. @stephenworsley changed tests to account for more detailed descriptions of projections in GDAL. (PR #3762) (see also GDAL#1185)
6. @stephenworsley changed tests to account for GDAL now saving fill values for data without masked points. (PR #3762)
7. @trexfeathers changed every graphics test that includes Cartopy's coastlines to account for new adaptive coastline scaling. (PR #3762) (see also Cartopy#1105)
8. @trexfeathers changed graphics tests to account for some new default grid-line spacing in Cartopy. (PR #3762) (see also Cartopy#1117)
9. @trexfeathers added additional acceptable graphics test targets to account for very minor changes in matplotlib version `3.3` (colormaps, fonts and axes borders). (PR #3762)
10. @rcomer corrected the matplotlib backend in Iris tests to ignore `matplotlib.rcdefaults`, instead the tests will **always** use `agg`. (PR #3846)
11. @bjlittle migrated the black support from `19.10b0` to `20.8b1`. (PR #3866)
12. @lbdreyer updated the CF standard name table to the latest version: v75. (PR #3867)
13. @bjlittle added **PEP 517** and **PEP 518** support for building and installing Iris, in particular to handle the PyKE package dependency. (PR #3812)
14. @bjlittle added metadata support for comparing *`attributes`* dictionaries that contain numpy arrays using xxHash, an extremely fast non-cryptographic hash algorithm, running at RAM speed limits.

15. @bjlittle added the `iris.tests.assertDictEqual` method to override `unittest.TestCase.assertDictEqual()` in order to cope with testing metadata *`attributes`* dictionary comparison where the value of a key may be a numpy array. (PR #3785)
16. @bjlittle added the *`get_logger()`* function for creating a generic `logging.Logger` with a `logging.StreamHandler` and custom `logging.Formatter`. (PR #3785)
17. @owena11 identified and optimised a bottleneck in `FieldsFile` header loading due to the use of `numpy.fromfile()`. (PR #3791)
18. @znicholls added a test for plotting with the label being taken from the unit's symbol, see `test_pcolormesh_str_symbol()` (PR #3902).
19. @znicholls made `step_over_diffs()` robust to hyphens (`-`) in the input path (i.e. the `result_dir` argument) (PR #3902).
20. @bjlittle migrated the CIaaS from travis-ci to cirrus-ci, and removed stickler-ci support. (PR #3928)
21. @bjlittle introduced nox as a common and easy entry-point for test automation. It can be used both from cirrus-ci in the cloud, and locally by the developer to run the Iris tests, the doc-tests, the gallery doc-tests, and lint Iris with flake8 and black. (PR #3928)

28.3 v2.4 (20 Feb 2020)

This document explains the changes made to Iris for this release (*View all changes*.)

28.3.1 Features

Last python 2 version of Iris

Iris 2.4 is a final extra release of Iris 2, which back-ports specific desired features from Iris 3 (not yet released).

The purpose of this is both to support early adoption of certain newer features, and to provide a final release for Python 2.

The next release of Iris will be version 3.0 : a major-version release which introduces breaking API and behavioural changes, and only supports Python 3.

- *`iris.coord_systems.Geostationary`* can now accept creation arguments of *false_easting=None* or *false_northing=None*, equivalent to values of 0. Previously these kwargs could be omitted, but could not be set to *None*. This also enables loading of netcdf data on a Geostationary grid, where either of these keys is not present as a grid-mapping variable property : Previously, loading any such data caused an exception.
- The area weights used when performing area weighted regridding with *`iris.analysis.AreaWeighted`* are now cached. This allows a significant speed up when regridding multiple similar cubes, by repeatedly using a *`iris.analysis.AreaWeighted.regridder()`* objects which you created first.
- Name constraint matching against cubes during loading or extracting has been relaxed from strictly matching against the *`name()`*, to matching against either the `standard_name`, `long_name`, NetCDF `var_name`, or `STASH` attributes metadata of a cube.
- Cubes and coordinates now have a new `names` property that contains a tuple of the `standard_name`, `long_name`, NetCDF `var_name`, and `STASH` attributes metadata.

- The *NameConstraint* provides richer name constraint matching when loading or extracting against cubes, by supporting a constraint against any combination of `standard_name`, `long_name`, NetCDF `var_name` and `STASH` from the attributes dictionary of a *Cube*.

28.3.2 Bugs Fixed

- Fixed a problem which was causing file loads to fetch *all* field data whenever UM files (PP or Fieldsfiles) were loaded. With large source files, initial file loads are slow, with large memory usage before any cube data is even fetched. Large enough files will cause a crash. The problem occurs only with Dask versions >= 2.0.

28.3.3 Internal

- Iris is now able to use the latest version of matplotlib.

28.4 v2.3 (19 Dec 2019)

This document explains the changes made to Iris for this release (*View all changes*.)

28.4.1 Features

Support for CF 1.7

We have introduced several changes that contribute to Iris's support for the CF Conventions, including some CF 1.7 additions. We are now able to support:

- *Climatological Coordinates*
- *Standard name modifiers*
- *Geostationary projection*

You can read more about each of these below.

Additionally, the conventions attribute, added by Iris when saving to NetCDF, has been updated to `CF-1.7`, accordingly.

Climatological Coordinate Support

Iris can now load, store and save NetCDF climatological coordinates. Any cube time coordinate can be marked as a climatological time axis using the boolean property: `climatological`. The climatological bounds are stored in the coordinate's `bounds` property.

When an Iris climatological coordinate is saved in NetCDF, the NetCDF coordinate variable will be given a 'climatology' attribute, and the contents of the coordinate's `bounds` property are written to a NetCDF boundary variable called '<coordinate-name>_bounds'. These are in place of a standard 'bounds' attribute and accompanying boundary variable. See below for an example adapted from CF conventions:

```
dimensions:
  time=4;
  bnds=2;
variables:
  float temperature(time,lat,lon);
```

(continues on next page)

(continued from previous page)

```
    temperature:long_name="surface air temperature";
    temperature:cell_methods="time: minimum within years time: mean over years
↪";
    temperature:units="K";
  double time(time);
    time:climatology="time_climatology";
    time:units="days since 1960-1-1";
  double time_climatology(time,bnds);
data:  // time coordinates translated to date/time format
  time="1960-4-16", "1960-7-16", "1960-10-16", "1961-1-16" ;
  time_climatology="1960-3-1",  "1990-6-1",
                   "1960-6-1",  "1990-9-1",
                   "1960-9-1",  "1990-12-1",
                   "1960-12-1", "1991-3-1" ;
```

If a climatological time axis is detected when loading NetCDF - indicated by the format described above - the `climatological` property of the Iris coordinate will be set to `True`.

New Chunking Strategy

Iris now makes better choices of Dask chunk sizes when loading from NetCDF files: If a file variable has small, specified chunks, Iris will now choose Dask chunks which are a multiple of these up to a default target size.

This is particularly relevant to files with an unlimited dimension, which previously could produce a large number of small chunks. This had an adverse effect on performance.

In addition, Iris now takes its default chunk size from the default configured in Dask itself, i.e. `dask.config.get('array.chunk-size')`.

Lazy Statistics

Several statistical operations can now be done lazily, taking advantage of the performance improvements offered by Dask:

- *aggregated_by()*
- *RMS* (more detail below)
- *MEAN*

- Cube data equality testing (and hence cube equality) now uses a more relaxed tolerance : This means that some cubes may now test 'equal' that previously did not. Previously, Iris compared cube data arrays using `abs(a - b) < 1.e-8`

  We now apply the default operation of `numpy.allclose()` instead, which is equivalent to `abs(a - b) < (1.e-8 + 1.e-5 * b)`
- Added support to render HTML for *CubeList* in Jupyter Notebooks and JupyterLab.
- Loading CellMeasures with integer values is now supported.
- New coordinate system: *iris.coord_systems.Geostationary*, including load and save support, based on the CF Geostationary projection definition.

- *iris.coord_systems.VerticalPerspective* can now be saved to and loaded from NetCDF files.
- *iris.experimental.regrid.PointInCell* moved to *iris.analysis.PointInCell* to make this regridding scheme public
- Iris now supports standard name modifiers. See Appendix C, Standard Name Modifiers for more information.
- *iris.cube.Cube.remove_cell_measure()* now also allows removal of a cell measure by its name (previously only accepted a CellMeasure object).
- The *iris.analysis.RMS* aggregator now supports a lazy calculation. However, the "weights" keyword is not currently supported by this, so a *weighted* calculation will still return a realised result, *and* force realisation of the original cube data.
- Iris now supports NetCDF Climate and Forecast (CF) Metadata Conventions 1.7 (see CF 1.7 Conventions Document for more information)
- Updated standard name support to CF standard name table version 70, 2019-12-10
- Updated UM STASH translations to metarelate/metOcean commit 448f2ef, 2019-11-29

28.4.2 Bugs Fixed

- Cube equality of boolean data is now handled correctly.
- Fixed a bug where cell measures were incorrect after a cube *transpose()* operation. Previously, this resulted in cell-measures that were no longer correctly mapped to the cube dimensions.
- The *AuxCoord* disregarded masked points and bounds, as did the *DimCoord*. Fix permits an *AuxCoord* to contain masked points/bounds, and a TypeError exception is now raised when attempting to create or set the points/bounds of a *DimCoord* with arrays with missing points.
- *iris.coord_systems.VerticalPerspective* coordinate system now uses the CF Vertical perspective definition; had been erroneously using Geostationary.
- *CellMethod* will now only use valid NetCDF name tokens to reference the coordinates involved in the statistical operation.
- The following var_name properties will now only allow valid NetCDF name tokens to reference the said NetCDF variable name. Note that names with a leading underscore are not permitted.
- *iris.aux_factory.AuxCoordFactory.var_name*
- *iris.coords.CellMeasure.var_name*
- *iris.coords.Coord.var_name*
- *iris.coords.AuxCoord.var_name*
- *iris.cube.Cube.var_name*
- Rendering a cube in Jupyter will no longer crash for a cube with attributes containing `\n`.
- NetCDF variables which reference themselves in their `cell_measures` attribute can now be read.
- *quiver()* now handles circular coordinates.
- The names of cubes loaded from abf/abl files have been corrected.
- Fixed a bug in UM file loading, where any landsea-mask-compressed fields (i.e. with LBPACK=x2x) would cause an error later, when realising the data.
- *iris.cube.Cube.collapsed()* now handles partial collapsing of multidimensional coordinates that have bounds.

- Fixed a bug in the *PROPORTION* aggregator, where cube data in the form of a masked array with `array.mask=False` would cause an error, but possibly only later when the values are actually realised. (Note: since netCDF4 version 1.4.0, this is now a common form for data loaded from netCDF files).
- Fixed a bug where plotting a cube with a *iris.coord_systems.LambertConformal* coordinate system would result in an error. This would happen if the coordinate system was defined with one standard parallel, rather than two. In these cases, a call to *as_cartopy_crs()* would fail.
- *iris.cube.Cube.aggregated_by()* now gives correct values in points and bounds when handling multidimensional coordinates.
- Fixed a bug in the *iris.cube.Cube.collapsed()* operation, which caused the unexpected realization of any attached auxiliary coordinates that were *bounded*. It now correctly produces a lazy result and does not realise the original attached AuxCoords.

28.4.3 Internal

- Iris now supports Proj4 up to version 5, but not yet 6 or beyond, pending fixes to some cartopy tests.
- Iris now requires Dask >= 1.2 to allow for improved coordinate equality checks.

28.4.4 Documentation

- Adopted a new colour logo for Iris
- Added a gallery example showing how to concatenate NEMO ocean model data, see *Load a Time Series of Data From the NEMO Model*.
- Added an example for loading Iris cubes for *Constraining on Time* in the user guide, demonstrating how to load data within a specified date range.
- Added notes to the *iris.load()* documentation, and the user guide *Loading Iris Cubes* chapter, emphasizing that the *order* of the cubes returned by an iris load operation is effectively random and unstable, and should not be relied on.
- Fixed references in the documentation of *iris.util.find_discontiguities()* to a non existent "mask_discontiguities" routine : these now refer to *mask_cube()*.

28.5 v2.2 (11 Oct 2018)

This document explains the changes made to Iris for this release (*View all changes*.)

28.5.1 Features

2-Dimensional Coordinate Plotting

The Iris plot functions *pcolor()* and *pcolormesh()* now accommodate the plotting of 2-dimensional coordinates as well as 1-dimensional coordinates.

To enable this feature, each coordinate passed in for plotting will be automatically checked for contiguity. Coordinate bounds must either be contiguous, or the cube's data must be masked at the discontiguities in order to avoid plotting errors.

The Iris plot function *iris.plot.quiver()* has been added, and this also works with 2-dimensional plot coordinates.

2-Dimensional Grid Vectors

The Iris functions *iris.analysis.cartography.gridcell_angles()* and *iris.analysis.cartography.rotate_grid_vectors()* have been added, allowing you to convert gridcell-oriented vectors to true-North/East ones.

NetCDF Data Variable Chunk Sizes

NetCDF data variable chunk sizes are now utilised at load time for significant performance improvements.

- The *iris.fileformats.um.FieldCollation* objects, which are passed into load callbacks when using *iris.fileformats.um.structured_um_loading()*, now have the additional properties: *iris.fileformats.um.FieldCollation.data_filepath* and *iris.fileformats.um.FieldCollation.data_field_indices*. These provide the file locations of the original data fields, which are otherwise lost in the structured loading process.
- *iris.util.reverse()* can now be used to reverse a cube by specifying one or more coordinates.
- Time mean fields can now be saved to PP files as a cell method.
- Cube aggregators *iris.analysis.MIN()*, *iris.analysis.MAX()*, *iris.analysis.SUM()* and *iris.analysis.COUNT()* now perform lazy aggregation by utilizing dask.
- Error messages thrown upon failed addition of an *AuxCoordFactory* now include the name of the required (but absent) coordinate as well as the name of the cube.
- The function *iris.util.find_discontiguities()* can be used to check for discontiguities in the bounds arrays of cube coordinates. Additionally, discontiguous points in coordinates can be explicitly masked using another new feature *iris.util.mask_cube()*.
- *iris.util.array_equal()* now has a 'withnans' keyword, which provides a NaN-tolerant array comparison.

28.5.2 Bugs Fixed

- The bug has been fixed that prevented printing time coordinates with bounds when the time coordinate was measured on a long interval (that is, `months` or `years`).
- "Gracefully filling…" warnings are now only issued when the coordinate or bound data is actually masked.

v2.2.1 (28 May 2019)

- Iris can now correctly unpack a column of header objects when saving a pandas DataFrame to a cube.
- fixed a bug in *iris.util.new_axis()* : copying the resulting cube resulted in an exception, if it contained an aux-factory.
- *iris.coords.AuxCoord*'s can now test as 'equal' even when the points or bounds arrays contain NaN values, if values are the same at all points. Previously this would fail, as conventionally "NaN != NaN" in normal floating-point arithmetic.

28.5.3 Internal

- Iris is now using the latest version release of dask (currently 0.19.3)
- Proj4 has been temporarily pinned to version < 5 while problems with the Mollweide projection are addressed.
- Matplotlib has been pinned to version < 3 temporarily while we account for its changes in all SciTools libraries.

28.5.4 Documentation

- Iris' *INSTALL* document has been updated to include guidance for running tests.
- A link has been added to the Developers' Guide to make it easier to find the Pull Request Check List.

28.6 v2.1 (06 Jun 2018)

This document explains the changes made to Iris for this release (*View all changes*.)

28.6.1 Features

- Added `repr_html` functionality to the *Cube* to provide a rich html representation of cubes in Jupyter notebooks. Existing functionality of `print(cube)` is maintained.

In [3]:
```
filename = iris.sample_data_path('rotated_pole.nc')
cube = iris.load_cube(filename)
cube
```

Out[3]:

Air Pressure At Sea Level (Pa)	grid_latitude	grid_longitude
Shape	22	36
Dimension coordinates		
grid_latitude	x	-
grid_longitude	-	x
Scalar coordinates		
forecast_period	0.0 hours	
forecast_reference_time	2006-06-15 00:00:00	
time	2006-06-15 00:00:00	
Attributes		
Conventions	CF-1.5	
STASH	m01s16i222	
source	Data from Met Office Unified Model 6.01	

- Updated *iris.cube.Cube.name()* to return a STASH code if the cube has one and no other valid names are present. This is now consistent with the summary information from *iris.cube.Cube.summary()*.
- The partial collapse of multi-dimensional auxiliary coordinates is now supported. Collapsed bounds span the range of the collapsed dimension(s).
- Added new function *iris.cube.CubeList.realise_data()* to compute multiple lazy values in a single operation, avoiding repeated re-loading of data or re-calculation of expressions.
- The methods *iris.cube.Cube.convert_units()* and *iris.coords.Coord.convert_units()* no longer forcibly realise the cube data or coordinate points/bounds. The converted values are now lazy arrays if the originals were.
- Added *iris.analysis.trajectory.interpolate()* that allows you to interpolate to find values along a trajectory.
- It is now possible to add an attribute of `missing_value` to a cube (Issue #1588).
- Iris can now represent data on the Albers Equal Area Projection, and the NetCDF loader and saver were updated to handle this. (Issue #2943)
- The *Mercator* projection has been updated to accept the `standard_parallel` keyword argument (PR #3041).

28.6.2 Bugs Fixed

- All var names being written to NetCDF are now CF compliant. Non alpha-numeric characters are replaced with '_', and var names now always have a leading letter (PR #2930).
- A cube resulting from a regrid operation using the *iris.analysis.AreaWeighted* regridding scheme will now have the smallest floating point data type to which the source cube's data type can be safely converted using NumPy's type promotion rules.
- `iris.quickplot` labels now honour the axes being drawn to when using the `axes` keyword (PR #3010).

28.6.3 Incompatible Changes

- The deprecated `iris.experimental.um` was removed. Please use consider using mule as an alternative.
- This release of Iris contains a number of updated metadata translations. See this changelist for further information.

28.6.4 Internal

- The cf_units dependency was updated to cf_units `v2.0`. cf_units v2 is almost entirely backwards compatible with v1. However the ability to preserve some aliased calendars has been removed. For this reason, it is possible that NetCDF load of a variable with a "standard" calendar will result in a saved NetCDF of a "gregorian" calendar.
- Iris updated its time-handling functionality from the netcdf4-python `netcdftime` implementation to the standalone module cftime. cftime is entirely compatible with netcdftime, but some issues may occur where users are constructing their own datetime objects. In this situation, simply replacing `netcdftime.datetime` with `cftime.datetime` should be sufficient.
- Iris now requires version 2 of Matplotlib, and `>=1.14` of NumPy. Full requirements can be seen in the requirements directory of the Iris' the source.

28.7 v2.0 (14 Feb 2018)

This document explains the changes made to Iris for this release (*View all changes*.)

28.7.1 Features

Dask Integration

The use of Biggus to provide support for *virtual arrays* and *lazy evaluation* within Iris has been replaced with Dask.

In addition the concept of *lazy data*, already used for the `Cube` data component, has now been extended to the data arrays of a `Coord` and an `AuxCoordFactory`.

This is a major feature enhancement, allowing Iris to leverage dask's rich functionality and community knowledge.

In particular, Dask's *threaded*, *multiprocessing* or *distributed* schedulers can be used in order to best utilise available compute and memory resource. For further details, see *Real and Lazy Data*.

- Changes to the `iris.cube.Cube`:
  - The *new* `core_data()` method returns the *real* or *lazy* `Cube` `data`.

  - The *new* in-place arithmetic operators `__iadd__`, `__idiv__`, `__imul__`, `__isub__`, and `__itruediv__` have been added to support *Cube* operations +=, /=, *=, and -=. Note that, for **divison** *__future__.division* is always in effect.

- Changes to the *iris.coords.Coord*:
  - The *new* *bounds_dtype* property (read-only) provides the `dtype` of the coordinate bounds, if they exist.
  - The *new* *core_points()* and *core_bounds()* methods return the *real* or *lazy* *Coord* `points` and `bounds` data, respectively.
  - The *new* *has_lazy_points()* and *has_lazy_bounds()* boolean methods return whether the coordinate has *lazy* `points` and *lazy* `bounds` data, respectively.
  - The *new* *lazy_points()* and *lazy_bounds()* methods return *lazy* representations of the coordinate `points` and `bounds` data, respectively.

The `iris.FUTURE` has Arrived!

Throughout version 1 of Iris a set of toggles in *iris.FUTURE* were maintained. These toggles allowed certain "future" behaviour to be enabled. Since the future has now arrived in Iris 2, all existing toggles in *iris.FUTURE* now default to `True`.

- `iris.Future.cell_datetime_objects`
  - Use of this FUTURE toggle is now deprecated.
  - *iris.coords.Cell* objects in time coordinates now contain datetime objects by default and not numbers. For example:

```
  >>> cube = iris.load_cube(iris.sample_data_path('air_temp.pp'))
  >>> print(cube.coord('time').cell(0).point)
      1998-12-01 00:00:00

This change particularly impacts constraining datasets on time. All time␣
↪constraints
must now be constructed with datetime objects or :class:`~iris.time.
↪PartialDateTime` objects.
See userguide section 2.2.1 for more details on producing time constraints.
```

- `iris.Future.netcdf_promote`
  - Use of this FUTURE toggle is now deprecated.
  - Removed deprecated behaviour that does not automatically promote NetCDF variables to cubes. This change means that NetCDF variables that define reference surfaces for dimensionless vertical coordinates will always be promoted and loaded as independent cubes.
- `iris.Future.netcdf_no_unlimited`
  - Use of this FUTURE toggle is now deprecated.
  - Removed deprecated behaviour that automatically set the length of the outer netCDF variable to 'UNLIMITED' on save. This change means that no cube dimension coordinates will be automatically saved as netCDF variables with 'UNLIMITED' length.
  - You can manually specify cube dimension coordinates to save with 'UNLIMITED' length. For example:

```
>>> iris.save(my_cube, 'my_result_file.nc', unlimited_dimensions=['latitude'])
```

- `iris.Future.clip_latitudes`

  - Use of this FUTURE toggle is now deprecated.
  - The *iris.coords.Coord.guess_bounds()* now limits the guessed bounds to [-90, 90] for latitudes by default. The ability to turn this behaviour off is now deprecated.

28.7.2 Bugs Fixed

- Indexing or slicing an *AuxCoord* coordinate will return a coordinate with `points` and `bounds` data that are new copied arrays, rather than views onto those of the original parent coordinate.
- Indexing or slicing a cell measure will return a new cell measure with `data` that is a new copied array, rather than a view onto the original parent cell measure.
- Performing an in-place arithmetic *add()*, *divide()*, *multiply()*, or *subtract()* operation on a *Cube* with `integer` or `boolean` data with a `float` result will raise an `ArithmeticError` exception.
- Lazy data now refers to absolute paths rather than preserving the form that was passed to iris.load functions. This means that it is possible to use relative paths at load, change working directory, and still expect to be able to load any un-loaded/lazy data. (#2325)
- The order in which files are passed to iris.load functions is now the order in which they are processed. (#2325)
- Loading from netCDF files with *iris.load()* will load a cube for each scalar variable, a variable that does not reference a netCDF dimension, unless that scalar variable is identified as a CF scalar coordinate, referenced from another data variable via the 'coordinates' attribute. Previously such data variables were ignored during load.

28.7.3 Incompatible Changes

- The *lazy_data()* method no longer accepts any arguments. Setting lazy data should now be done with *cube.data*.

Significant Changes in Calculated Results

Due to the replacement of Biggus with Dask, as described above, the results of certain types of calculation may have significantly different values from those obtained in earlier versions. This is of a much greater order than the usual small changes in floating point results : it applies especially to any data with masked points, or of long integer types.

- Due to concerns regarding maintainability and API consistency the `iris.cube.Cube.share_data` flag introduced in v1.13 has been removed. Intra-cube data sharing is a oft-requested feature that we will be targeting in a future Iris release.
- Using *convert_units()* on a cube with unknown units will now result in a `UnitConversionError` being raised.
- `iris.fileformats.pp_rules` has been renamed to *iris.fileformats.pp_load_rules* for consistency with the new *iris.fileformats.pp_save_rules*.
- Fill values are no longer taken from the cube's *data* attribute when it is a masked array.
- When saving a cube or list of cubes in NetCDF format, a fill value or list of fill values can be specified via a new *fill_value* argument. If a list is supplied, each fill value will be applied to each cube in turn. If a *fill_value* argument is not specified, the default fill value for the file format and the cube's data type will be used.
- When saving to PP, the "standard" BMDI of -1e30 is now always applied in `PPField` generation. To save PP data with an alternative BMDI, use *iris.fileformats.pp.save_pairs_from_cube()* to generate `PPFields`, and modify these before saving them to file.

- A 'fill_value' key can no longer be specified as part of the *packing* argument to *iris.save* when saving in netCDF format. Instead, a fill value or list of fill values should be specified as a separate *fill_value* argument if required.
- If the *packing* argument to *iris.save* is a dictionary, an error is raised if it contains any keys other than 'dtype', 'scale_factor' and 'add_offset'.
- The deprecated `iris.fileformats.grib` was removed. All Iris GRIB functionality is now delivered through `iris-grib`.
- In Iris v1 it was possible to configure Iris to log at import time through *iris.config.LOGGING*. This capability has been removed in Iris v2.
- When coordinates have no well defined plot axis, *iris.plot* and *iris.quickplot* routines now use the order of the cube's dimensions to determine the coordinates to plot as the x and y axis of a plot.
- The cf_units dependency version has been updated to v1.2.0, which prints shorter unit strings. For example, the unit `meter-second^-1` is now printed as `m.s-1`.

28.7.4 Deprecation

All deprecated functionality that was announced for removal in Iris 2.0 has been removed. In particular:

- The deprecated keyword arguments `coord` and `name` have been removed from the *iris.cube.Cube* constructor.
- The deprecated methods `iris.cube.Cube.add_history`, `iris.cube.Cube.assert_valid` and `iris.cube.Cube.regridded` have been removed from *iris.cube.Cube*.
- The deprecated module `iris.fileformats.pp_packing` has been removed.
- The deprecated module `iris.proxy` has been removed.
- The deprecated configuration variable `SAMPLE_DATA_DIR` has been removed from `iris.config` in favour of user installation of the `iris-sample-data` package.
- The deprecated module `iris.unit` has been removed in favour of `cf_units`.
- The `BitwiseInt` class has been removed from *iris.fileformats.pp*.
- Removed deprecated functions `reset_load_rules`, `add_save_rules`, `reset_save_rules` and `as_pairs` from *iris.fileformats.pp*.
- The deprecated module `iris.analysis.interpolate` has been removed, along with the following deprecated classes and functions:
  - `iris.analysis.interpolate.linear`
  - `iris.analysis.interpolate.nearest_neighbour_data_value`
  - `iris.analysis.interpolate.regrid`
  - `iris.analysis.interpolate.regrid_to_max_resolution`
  - `iris.analysis.interpolate.extract_nearest_neighbour`
  - `iris.analysis.interpolate.nearest_neighbour_indices`
  - `iris.analysis.interpolate.Linear1dExtrapolator`
- Removed deprecated module `iris.experimental.fieldsfile`. Note that there is no direct replacement for `:meth:iris.experimental.fieldsfile.load`, which specifically performed fast loading from _either_ PP or FF files. Instead, please use the *:meth:iris.fileformats.um.structured_um_loading* context manager, and within that context call *:meth:iris.load*, or the format-specific *:meth:iris.fileformats.pp.load_cubes* or *:meth:iris.fileformats.um.load_cubes*.
- Removed deprecated module `iris.fileformats.ff`. Please use facilities in *iris.fileformats.um* instead.

- **Removed deprecated and unused kwarg `ignore` from the following functions:**
  - *iris.analysis.calculus.curl()*,
  - *iris.analysis.maths.add()*, and
  - *iris.analysis.maths.subtract()*.
- Deprecated functions `iris.util.broadcast_weights`, `iris.util.ensure_array` and `iris.util.timers` have been removed from *iris.util*.
- The following classes and functions have been removed from *iris.fileformats.rules*:
  - `iris.fileformat.rules.calculate_forecast_period`
  - `iris.fileformat.rules.log`
  - `iris.fileformat.rules.CMAttribute`
  - `iris.fileformat.rules.CMCustomAttribute`
  - `iris.fileformat.rules.CoordAndDims`
  - `iris.fileformat.rules.DebugString`
  - `iris.fileformat.rules.FunctionRule`
  - `iris.fileformat.rules.ProcedureRule`
  - `iris.fileformat.rules.Rule`
  - `iris.fileformat.rules.RulesContainer`
  - `iris.fileformat.rules.RuleResult`
- In addition the deprecated keyword argument `legacy_custom_rules` has been removed from the *iris.fileformats.rules.Loader* constructor.

28.7.5 Documentation

- A new UserGuide chapter on *Real and Lazy Data* has been added, and referenced from key points in the *User Guide* .

28.8 v1.13 (17 May 2017)

This document explains the changes made to Iris for this release (*View all changes*.)

28.8.1 Features

- Allow the reading of NAME trajectories stored by time instead of by particle number.
- An experimental link to python-stratify via *iris.experimental.stratify*.
- Data arrays may be shared between cubes, and subsets of cubes, by using the `iris.cube.share_data()` flag.

28.8.2 Bug Fixes

- The bounds are now set correctly on the longitude coordinate if a zonal mean diagnostic has been loaded from a PP file as per the CF Standard.
- NetCDF loading will now determine whether there is a string-valued scalar label, i.e. a character variable that only has one dimension (the length of the string), and interpret this correctly.
- A line plot of geographic coordinates (e.g. drawing a trajectory) wraps around the edge of the map cleanly, rather than plotting a segment straight across the map.
- When saving to PP, lazy data is preserved when generating PP fields from cubes so that a list of cubes can be saved to PP without excessive memory requirements.
- An error is now correctly raised if a user tries to perform an arithmetic operation on two cubes with mismatching coordinates. Previously these cases were caught by the add and subtract operators, and now it is also caught by the multiply and divide operators.
- Limited area Rotated Pole datasets where the data range is `0 <= lambda < 360`, for example as produced in New Zealand, are plotted over a sensible map extent by default.
- Removed the potential for a RuntimeWarning: overflow encountered in `int_scalars` which was missed during collapsed calculations. This could trip up unwary users of limited data types, such as int32 for very large numbers (e.g. seconds since 1970).
- The CF conventions state that certain `formula_terms` terms may be omitted and assumed to be zero (http://cfconventions.org/cf-conventions/v1.6.0/cf-conventions.html#dimensionless-v-coord) so Iris now allows factories to be constructed with missing terms.
- In the User Guide's contour plot example, clabel inline is set to be False so that it renders correctly, avoiding spurious horizontal lines across plots, although this does make labels a little harder to see.
- The computation of area weights has been changed to a more numerically stable form. The previous form converted latitude to colatitude and used difference of cosines in the cell area computation. This formulation uses latitude and difference of sines. The conversion from latitude to colatitude at lower precision causes errors when computing the cell areas.

28.8.3 Testing

- Iris has adopted conda-forge to provide environments for continuous integration testing.

28.9 v1.12 (31 Jan 2017)

This document explains the changes made to Iris for this release (*View all changes*.)

28.9.1 Features

Showcase Feature: New regridding schemes

A new regridding scheme, *iris.analysis.UnstructuredNearest*, performs nearest-neighbour regridding from "unstructured" onto "structured" grids. Here, "unstructured" means that the data has X and Y coordinate values defined at each horizontal location, instead of the independent X and Y dimensions that constitute a structured grid. For example, data sampled on a trajectory or a tripolar ocean grid would be unstructured.

In addition, added experimental ProjectedUnstructured regridders which use scipy.interpolate.griddata to regrid unstructured data (see *iris.experimental.regrid.ProjectedUnstructuredLinear* and *iris.experimental.regrid.ProjectedUnstructuredNearest*). The essential purpose is the same as *iris.analysis.UnstructuredNearest*. This scheme, by comparison, is generally faster, but less accurate.

Showcase Feature: Fast UM file loading

Support has been added for accelerated loading of UM files (PP and Fieldsfile), when these have a suitable regular "structured" form.

A context manager is used to enable fast um loading in all the regular Iris load functions, such as *iris.load()* and *iris.load_cube()*, when loading data from UM file types. For example:

```
>>> import iris
>>> filepath = iris.sample_data_path('uk_hires.pp')
>>> from iris.fileformats.um import structured_um_loading
>>> with structured_um_loading():
...     cube = iris.load_cube(filepath, 'air_potential_temperature')
```

This approach can deliver loading which is 10 times faster or more. For example :

- a 78 Gb fieldsfile of 51,840 fields loads in about 13 rather than 190 seconds.
- a set of 25 800Mb PP files loads in about 21 rather than 220 seconds.

You can load data with structured loading and compare the results with those from "normal" loading to check whether they are equivalent.

- The results will normally differ, if at all, only in having dimensions in a different order or a different choice of dimension coordinates. **In these cases, structured loading can be used with confidence.**
- Ordinary Fieldsfiles (i.e. model outputs) are generally suitable for structured loading. Many PP files also are, especially if produced directly from Fieldsfiles, and retaining the same field ordering.
- Some inputs however (generally PP) will be unsuitable for structured loading : For instance if a particular combination of vertical levels and time has been omitted, or some fields appear out of order.
- There are also some known unsupported cases, including data which is produced on pseudo-levels. See the detail documentation on this.

It is the user's responsibility to use structured loading only with suitable inputs. Otherwise, odd behaviour and even incorrect loading can result, as input files are not checked as fully as in a normal load.

Although the user loading call for structured loading can be just the same, and the returned results are also often identical, structured loading is not in fact an exact *identical* replacement for normal loading:

- results are often somewhat different, especially regarding the order of dimensions and the choice of dimension coordinates.
- although both constraints and user callbacks are supported, callback routines will generally need to be re-written. This is because a 'raw' cube in structured loading generally covers *multiple* PPfields, which therefore need to be handled as a collection : A grouping object containing them is passed to the callback 'field' argument. An example showing callbacks suitable for both normal and structured loading can be seen here.

For full details, see : *iris.fileformats.um.structured_um_loading()*.

- A skip pattern is introduced to the fields file loader, such that fields which cannot be turned into iris PPField instances are skipped and the remaining fields are loaded. This especially applies to certain types of files that

can contain fields with a non-standard LBREL value : Iris can now load such a file, skipping the unreadable field and printing a warning message.

- Iris can now load PP files containing a PP field whose LBLREC value does not match the field length recorded in the file. A warning message is printed, and all fields up to the offending one are loaded and returned. Previously, this simply resulted in an unrecoverable error.
- The transpose method of a Cube now results in a lazy transposed view of the original rather than realising the data then transposing it.
- The *iris.analysis.cartography.area_weights()* function is now more accurate for single precision input bounds.
- Iris is now able to read seconds in datetimes provided in NAME trajectory files.
- Optimisations to trajectory interpolations have resulted in a significant speed improvement.
- Many new and updated translations between CF spec and STASH codes.

28.9.2 Deprecations

- The module `iris.experimental.fieldsfile` has been deprecated, in favour of the new fast-loading mechanism provided by *iris.fileformats.um.structured_um_loading()*.

28.9.3 Documentation

- Corrected documentation of *iris.analysis.AreaWeighted* scheme to make the usage scope clearer.

28.10 v1.11 (29 Oct 2016)

This document explains the changes made to Iris for this release (*View all changes*.)

28.10.1 Features

- If available, display the `STASH` code instead of `unknown / (unknown)` when printing cubes with no `standard_name` and no `units`.
- Support for saving to netCDF with data packing has been added.
- The coordinate system *iris.coord_systems.LambertAzimuthalEqualArea* has been added with NetCDF saving support.

28.10.2 Bugs Fixed

- Fixed a floating point tolerance bug in *iris.experimental.regrid.regrid_area_weighted_rectilinear_src_and_grid()* for wrapped longitudes.
- Allow *iris.util.new_axis()* to promote the nominated scalar coordinate of a cube with a scalar masked constant data payload.
- Fixed a bug where `iris.util._is_circular()` would erroneously return false when coordinate values are decreasing.

- When saving to NetCDF, the existing behaviour of writing string attributes as ASCII has been maintained across known versions of netCDF4-python.

28.10.3 Documentation

- Fuller doc-string detail added to *iris.analysis.cartography.unrotate_pole()* and *iris.analysis.cartography.rotate_pole()*.

28.11 v1.10 (05 Sep 2016)

This document explains the changes made to Iris for this release (*View all changes*.)

28.11.1 Features

- Support has now been added for the iris_grib package, which provides GRIB format support in an optional package, separate from Iris.
  - If `iris_grib` is available, it will always be used in place of the older iris module `iris.fileformats.grib`.
  - The capabilities of `iris_grib` are essentially the same as the existing `iris.fileformats.grib` when used with `iris.FUTURE.strict_grib_load=True`, with only small detail differences.
  - The old `iris.fileformats.grib` module is now deprecated and may shortly be removed.
    - If you are already using the recommended *iris.FUTURE* setting `iris.FUTURE.strict_grib_load=True` this should not cause problems, as the new package is all-but identical.
      - However, the option `iris.FUTURE.strict_grib_load` is itself now deprecated, so you should remove code that sets it.
    - If, however, your code is still using the older "non-strict" grib loading, then you may need to make code changes.
      - In particular, the `field` object passed to load callbacks is different. See `iris.fileformats.grib.message.GribMessage` (the `iris_grib.message.GribMessage` class is the same as this, for now).
  - Please exercise your code with the new iris_grib module, and let us know of any problems you uncover, such as files that will no longer load with the new implementation.
- `iris.experimental.regrid.PointInCell.regridder()` now works across coordinate systems, including non latlon systems. Additionally, the requirement that the source data X and Y coordinates be 2D has been removed. NB: some aspects of this change are backwards incompatible.
- Plotting non-Gregorian calendars is now supported. This adds nc_time_axis as a dependency.
- Promoting a scalar coordinate to a dimension coordinate with *iris.util.new_axis()* no longer loads deferred data.
- The parsing functionality for Cell Methods from netCDF files is available as part of the *iris.fileformats.netcdf* module as *iris.fileformats.netcdf.parse_cell_methods()*.
- Support for the NameIII Version 2 file format has been added.

- Loading netcdf data in Mercator and Stereographic projections now accepts optional extra projection parameter attributes (`false_easting`, `false_northing` and `scale_factor_at_projection_origin`), if they match the default values.
  - NetCDF files which define a Mercator projection where the `false_easting`, `false_northing` and `scale_factor_at_projection_origin` match the defaults will have the projection loaded correctly. Otherwise, a warning will be issued for each parameter that does not match the default and the projection will not be loaded.
  - NetCDF files which define a Steroegraphic projection where the `scale_factor_at_projection_origin` is equal to 1.0 will have the projection loaded correctly. Otherwise, a warning will be issued and the projection will not be loaded.
- The *iris.plot* routines *contour()*, *contourf()*, *outline()*, *pcolor()*, *pcolormesh()* and *points()* now support plotting cubes with anonymous dimensions by specifying the *numeric index* of the anonymous dimension within the `coords` keyword argument.

  Note that the axis of the anonymous dimension will be plotted in index space.
- NetCDF loading and saving now supports Cubes that use the LambertConformal coordinate system.
- The experimental structured Fieldsfile loader `load()` has been extended to also load structured PP files.

  Structured loading is a streamlined operation, offering the benefit of a significantly faster loading alternative to the more generic *iris.load()* mechanism.

  Note that structured loading is not an optimised wholesale replacement of *iris.load()*. Structured loading is restricted to input containing contiguously ordered fields for each phenomenon that repeat regularly over the same vertical levels and times. For further details, see `load()`
- *iris.experimental.regrid_conservative* is now compatible with ESMPy v7.
- Saving zonal (i.e. longitudinal) means to PP files now sets the '64s' bit in LBPROC.
- Loading of 'little-endian' PP files is now supported.
- All appropriate *iris.plot* functions now handle an `axes` keyword, allowing use of the object oriented matplotlib interface rather than pyplot.
- The ability to pass file format object lists into the rules based load pipeline, as used for GRIB, Fields Files and PP has been added. The *iris.fileformats.pp.load_pairs_from_fields()* and `iris.fileformats.grib.load_pairs_from_fields()` are provided to produce cubes from such lists. These lists may have been filtered or altered using the appropriate *iris.fileformats* modules.
- Cubes can now have an 'hour' coordinate added with *iris.coord_categorisation.add_hour()*.
- Time coordinates from PP fields with an lbcode of the form 3xx23 are now correctly encoded with a 360-day calendar.
- The loading from and saving to netCDF of CF cell_measure variables is supported, along with their representation within a Cube as *cell_measures*.
- Cubes with anonymous dimensions can now be concatenated. This can only occur along a dimension that is not anonymous.
- NetCDF saving of `valid_range`, `valid_min` and `valid_max` cube attributes is now allowed.

28.11.2 Bugs Fixed

- Altered Cell Methods to display coordinate's standard_name rather than var_name where appropriate to avoid human confusion.
- Saving multiple cubes with netCDF4 protected attributes should now work as expected.
- Concatenating cubes with singleton dimensions (dimensions of size one) now works properly.
- Fixed the `grid_mapping_name` and `secant_latitudes` handling for the LambertConformal coordinate system.
- Fixed bug in *iris.analysis.cartography.project()* where the output projection coordinates didn't have units.
- Attempting to use *iris.sample_data_path()* to access a file that isn't actually Iris sample data now raises a more descriptive error. A note about the appropriate use of *sample_data_path* has also been added to the documentation.
- Fixed a bug where regridding or interpolation with the *Nearest* scheme returned floating-point results even when the source data was integer typed. It now always returns the same type as the source data.
- Fixed a bug where regridding circular data would ignore any source masking. This affected any regridding using the *Linear* and *Nearest* schemes, and also `iris.analysis.interpolate.linear()`.
- The `coord_name` parameter to *scalar_cell_method()* is now checked correctly.
- LBPROC is set correctly when a cube containing the minimum of a variable is saved to a PP file. The IA component of LBTIM is set correctly when saving maximum or minimum values.
- The performance of *iris.cube.Cube.extract()* when a list of values is given to an instance of *iris.Constraint* has been improved considerably.
- Fixed a bug with *iris.cube.Cube.data()* where an `numpy.ndarray` was not being returned for scalar cubes with lazy data.
- When saving in netcdf format, the units of 'latitude' and 'longitude' coordinates specified in 'degrees' are saved as 'degrees_north' and 'degrees_east' respectively, as defined in the CF conventions for netCDF files: sections 4.1 and 4.2.
- Fixed a bug with a class of pp files with lbyr == 0, where the date would cause errors when converting to a datetime object (e.g. when printing a cube).

  When processing a pp field with lbtim = 2x, lbyr == lbyrd == 0 and lbmon == lbmond, 'month' and 'month_number' coordinates are created instead of 'time'.
- Fixed a bug in *curl()* where the sign of the r-component for spherical coordinates was opposite to what was expected.
- A bug that prevented cube printing in some cases has been fixed.
- Fixed a bug where a deepcopy of a *DimCoord* would have writeable `points` and `bounds` arrays. These arrays can now no longer be modified in-place.
- Concatenation no longer occurs when the auxiliary coordinates of the cubes do not match. This check is not applied to AuxCoords that span the dimension the concatenation is occurring along. This behaviour can be switched off by setting the `check_aux_coords` kwarg in *iris.cube.CubeList.concatenate()* to False.
- Fixed a bug in *iris.cube.Cube.subset()* where an exception would be thrown while trying to subset over a non-dimensional scalar coordinate.

28.11.3 Incompatible Changes

- The source and target for `iris.experimental.regrid.PointInCell.regridder()` must now have defined coordinate systems (i.e. not `None`). Additionally, the source data X and Y coordinates must have the same cube dimensions.

28.11.4 Deprecations

- Deprecated the *iris.Future* option `iris.FUTURE.strict_grib_load`. This only affected the module `iris.fileformats.grib`, which is itself now deprecated. Please see *iris_grib package*, above.
- Deprecated the module `iris.fileformats.grib`. The new package iris_grib replaces this functionality, which will shortly be removed. Please see *iris_grib package*, above.
- The use of `iris.config.SAMPLE_DATA_DIR` has been deprecated and replaced by the now importable iris_sample_data package.
- Deprecated the module `iris.analysis.interpolate`. This contains the following public items, all of which are now deprecated and will be removed in a future release:
  - `linear()`
  - `regrid()`
  - `regrid_to_max_resolution()`
  - `nearest_neighbour_indices()`
  - `nearest_neighbour_data_value()`
  - `extract_nearest_neighbour()`
  - class `Linear1dExtrapolator`.

  Please use the replacement facilities individually noted in the module documentation for `iris.analysis.interpolate`
- The method `iris.cube.Cube.regridded()` has been deprecated. Please use *iris.cube.Cube.regrid()* instead (see `regridded()` for details).
- Deprecated `iris.fileformats.grib.hindcast_workaround` and `iris.fileformats.grib.GribWrapper`. The class `iris.fileformats.grib.message.GribMessage` provides alternative means of working with GRIB message instances.
- Deprecated the module `iris.fileformats.ff`. Please use the replacement facilities in module *iris.fileformats.um* :
  - *iris.fileformats.um.um_to_pp()* replaces `iris.fileformats.ff.FF2PP`.
  - *iris.fileformats.um.load_cubes()* replaces `iris.fileformats.ff.load_cubes()`.
  - *iris.fileformats.um.load_cubes_32bit_ieee()* replaces `iris.fileformats.ff.load_cubes_32bit_ieee()`.

  All other public components are generally deprecated and will be removed in a future release.
- The `iris.fileformats.pp.as_pairs()` and `iris.fileformats.grib.as_pairs()` are deprecated. These are replaced with *iris.fileformats.pp.save_pairs_from_cube()* and `iris.fileformats.grib.save_pairs_from_cube()`.
- `iris.fileformats.pp_packing` has been deprecated. Please install the separate mo_pack package instead. This provides the same functionality.

- Deprecated logging functions (currently used only for rules logging): `iris.config.iris.config.RULE_LOG_DIR`, `iris.config.iris.config.RULE_LOG_IGNORE` and `iris.fileformats.rules.log`.
- Deprecated all the remaining text rules mechanisms: `iris.fileformats.rules.DebugString`, `iris.fileformats.rules.CMAttribute`, `iris.fileformats.rules.CMCustomAttribute`, `iris.fileformats.rules.CoordAndDims`, `iris.fileformats.rules.Rule`, `iris.fileformats.rules.FunctionRule`, `iris.fileformats.rules.ProcedureRule`, `iris.fileformats.rules.RulesContainer` and `iris.fileformats.rules.calculate_forecast_period()`.
- Deprecated the custom pp save rules mechanism implemented by the functions `iris.fileformats.pp.add_save_rules()` and `iris.fileformats.pp.reset_save_rules()`. The functions *iris.fileformats.pp.as_fields()*, `iris.fileformats.pp.as_pairs()` and *iris.fileformats.pp.save_fields()* provide alternative means of achieving the same ends.

28.11.5 Documentation

- It is now clear that repeated values will form a group under *iris.cube.Cube.aggregated_by()* even if they aren't consecutive. Hence, the documentation for *iris.cube* has been changed to reflect this.
- The documentation for *iris.analysis.calculus.curl()* has been updated for clarity.
- False claims about *iris.fileformats.pp.save()*, `iris.fileformats.pp.as_pairs()`, and *iris.fileformats.pp.as_fields()* being able to take instances of *iris.cube.CubeList* as inputs have been removed.
- A new code example *Plotting Wind Direction Using Quiver*, demonstrating the use of a quiver plot to display wind speeds over Lake Victoria, has been added.
- The docstring for *iris.analysis.SUM* has been updated to explicitly state that weights passed to it aren't normalised internally.
- A note regarding the impossibility of partially collapsing multi-dimensional coordinates has been added to the user guide.

28.12 v1.9 (10 Dec 2015)

This document explains the changes made to Iris for this release (*View all changes*.)

28.12.1 Features

- Support for running on Python 3.4 has been added to the whole code base. Some features which depend on external libraries will not be available until they also support Python 3, namely:
  - gribapi does not yet provide a Python 3 interface
- Added the UM pseudo level type to the information made available in the STASH_TRANS table in `iris.fileformats.um._ff_cross_references`
- When reading "cell_methods" attributes from NetCDF files, allow optional whitespace before the colon. This is not strictly in the CF spec, but is a common occurrence.
- Basic cube arithmetic (plus, minus, times, divide) now supports lazy evaluation.
- *iris.analysis.cartography.rotate_winds()* can now operate much faster on multi-layer (i.e. > 2-dimensional) cubes, as it calculates rotation coefficients only once and reuses them for additional layers.

- Linear regridding of a multi-layer (i.e. > 2-dimensional) cube is now much faster, as it calculates transform coefficients just once and reuses them for additional layers.
- Ensemble statistics can now be saved to GRIB2, using Product Definition Template 4.11.
- Loading of NetCDF data with ocean vertical coordinates now returns a 'depth' in addition to an 'eta' cube. This operates on specific defined dimensionless coordinates : see CF spec version 1.6, Appendix D.
- *iris.analysis.stats.pearsonr()* updates:
  - Cubes can now be different shapes, provided one is broadcastable to the other.
  - Accepts weights keyword for weighted correlations.
  - Accepts mdtol keyword for missing data tolerance level.
  - Accepts common_mask keyword for restricting calculation to unmasked pairs of cells.
- Added a new point-in-cell regridding scheme, *iris.experimental.regrid.PointInCell*.
- Added *iris.analysis.WPERCENTILE()* - a new weighted aggregator for calculating percentiles.
- Added cell-method translations for LBPROC=64 and 192 in UM files, encoding 'zonal mean' and 'zonal+time mean'.
- Support for loading GRIB2 messages defined on a Lambert conformal grid has been added to the GRIB2 loader.
- Data on potential-temperature (theta) levels can now be saved to GRIB2, with a fixed surface type of 107.
- Added several new helper functions for file-save customisation, (see also : *Saving Iris Cubes*):
  - `iris.fileformats.grib.as_pairs()`
  - `iris.fileformats.grib.as_messages()`
  - `iris.fileformats.grib.save_messages()`
  - `iris.fileformats.pp.as_pairs()`
  - *iris.fileformats.pp.as_fields()*
  - *iris.fileformats.pp.save_fields()*
- Loading data from GRIB2 now supports most of the currently defined 'data representation templates' : code numbers 0, 1, 2, 3, 4, 40, 41, 50, 51 and 61.
- When a Fieldsfile is opened for update as a `iris.experimental.um.FieldsFileVariant`, unmodified packed data in the file can now be retained in the original form. Previously it could only be stored in an unpacked form.
- When reading and writing NetCDF data, the CF 'flag' attributes, "flag_masks", "flag_meanings" and "flag_values" are now preserved through Iris load and save.
- mo_pack was added as an optional dependency. It is used to encode and decode data in WGDOS packed form.
- The `iris.experimental.um.Field.get_data()` method can now be used to read Fieldsfile data after the original `iris.experimental.um.FieldsFileVariant` has been closed.

28.12.2 Bugs Fixed

- Fixed a bug in `iris.unit.Unit.convert()` (and the equivalent in cf_units) so that it now converts data to the native endianness, without which udunits could not read it correctly.
- Fixed a bug with loading WGDOS packed data in `iris.experimental.um`, which could occasionally crash, with some data.
- Ignore non-numeric suffices in the numpy version string, which would otherwise crash some regridding routines.
- fixed a bug in *iris.fileformats.um_cf_map* where the standard name for the stash code m01s12i187 was incorrectly set, such that it is inconsistent with the stated unit of measure, 'm s-1'. The different name, a long_name of 'change_over_time_in_upward_air_velocity_due_to_advection' with units of 'm s-1' is now used instead.
- Fixed a bug in *iris.cube.Cube.intersection()*. When edge points were at (base + period), intersection would unnecessarily wrap the data.
- Fixed a bug in *iris.fileformats.pp*. A previous release removed the ability to pass a partial constraint on STASH attribute.
- *iris.plot.default_projection_extent()* now correctly raises an exception if a cube has X bounds but no Y bounds, or vice versa. Previously it never failed this, as the test was wrong.
- When loading NetCDF data, a "units" attribute containing unicode characters is now transformed by backslash-replacement. Previously this caused a crash. Note: unicode units are *not supported in the CF conventions*.
- When saving to NetCDF, factory-derived auxiliary coordinates are now correctly saved with different names when they are not identical. Previously, such coordinates could be saved with the same name, leading to errors.
- Fixed a bug in `iris.experimental.um.FieldsFileVariant.close()`, which now correctly allocates extra blocks for larger lookups when saving. Previously, when larger files open for update were closed, they could be written out with data overlapping the lookup table.
- Fixed a bug in *iris.aux_factory.OceanSigmaZFactory* which sometimes caused crashes when fetching the points of an "ocean sigma z" coordinate.

v1.9.1 (05 Jan 2016)

- Fixed a unicode bug preventing standard names from being built cleanly when installing in Python3

v1.9.2 (28 Jan 2016)

- New warning regarding data loss if writing to an open file which is also open to read, with lazy data.
- Removal of a warning about data payload loading from concatenate.
- Updates to concatenate documentation.
- Fixed a bug with a name change in the netcdf4-python package.
- Fixed a bug building the documentation examples.
- Fixed a bug avoiding sorting classes directly when *iris.cube.Cube.coord_system()* is used in Python3.
- Fixed a bug regarding unsuccessful dot import.

28.12.3 Incompatible Changes

- GRIB message/file reading and writing may not be available for Python 3 due to GRIB API limitations.

28.12.4 Deprecations

- Deprecated `iris.unit`, with unit functionality provided by cf_units instead.
- When loading from NetCDF, a deprecation warning is emitted if there is vertical coordinate information that *would* produce extra result cubes if `iris.FUTURE.netcdf_promote` were set, but it is *not* set.
- Deprecated `iris.aux_factory.LazyArray`

28.12.5 Documentation

- A chapter on *saving iris cubes* has been added to the *user guide*.
- Added script and documentation for building a what's new page from developer-submitted contributions. See *Contributing a "What's New" entry*.

28.13 v1.8 (14 Apr 2015)

This document explains the changes made to Iris for this release (*View all changes*.)

28.13.1 Features

Showcase: Rotate winds

Iris can now rotate and unrotate wind vector data by transforming the wind vector data to another coordinate system.

For example:

```
>>> from iris.analysis.cartography import rotate_winds
>>> u_cube = iris.load_cube('my_rotated_u_wind_cube.pp')
>>> v_cube = iris.load_cube('my_rotated_v_wind_cube.pp')
>>> target_cs = iris.coord_systems.GeogCS(6371229.0)
>>> u_prime, v_prime = rotate_winds(u_cube, v_cube, target_cs)
```

Showcase: Nearest-neighbour scheme

A nearest-neighbour scheme for interpolation and regridding has been added to Iris. This joins the existing *Linear* and *AreaWeighted* interpolation and regridding schemes.

For example:

```
>>> result = cube.interpolate(sample_points, iris.analysis.Nearest())
>>> regridded_cube = cube.regrid(target_grid, iris.analysis.Nearest())
```

Showcase: Slices over a coordinate

You can slice over one or more dimensions of a cube using *iris.cube.Cube.slices_over()*. This provides similar functionality to *slices()* but with almost the opposite outcome.

Using *slices()* to slice a cube on a selected dimension returns all possible slices of the cube with the selected dimension retaining its dimensionality. Using *slices_over()* to slice a cube on a selected dimension returns all possible slices of the cube over the selected dimension.

To demonstrate this:

```
>>> cube = iris.load(iris.sample_data_path('colpex.pp'))[0]
>>> print(cube.summary(shorten=True))
air_potential_temperature / (K)     (time: 6; model_level_number: 10; grid_latitude: 83;␣
↪grid_longitude: 83)
>>> my_slice = next(cube.slices('time'))
>>> my_slice_over = next(cube.slices_over('time'))
>>> print(my_slice.summary(shorten=True))
air_potential_temperature / (K)     (time: 6)
>>> print(my_slice_over.summary(shorten=True))
air_potential_temperature / (K)     (model_level_number: 10; grid_latitude: 83; grid_
↪longitude: 83)
```

- *iris.cube.CubeList.concatenate()* now works with biggus arrays and so now supports concatenation of cubes with deferred data.
- Improvements to NetCDF saving through using biggus:
- A cube's lazy data payload will still be lazy after saving; the data will not be loaded into memory by the save operation.
- Cubes with data payloads larger than system memory can now be saved to NetCDF through biggus streaming the data to disk.
- *iris.util.demote_dim_coord_to_aux_coord()* and *iris.util.promote_aux_coord_to_dim_coord()* allow a coordinate to be easily demoted or promoted within a cube.
- *iris.util.squeeze()* removes all length 1 dimensions from a cube, and demotes any associated squeeze dimension *DimCoord* to be a scalar coordinate.
- *iris.cube.Cube.slices_over()*, which returns an iterator of all sub-cubes along a given coordinate or dimension index.
- *iris.cube.Cube.interpolate()* now accepts datetime.datetime and netcdftime.datetime instances for date or time coordinates.
- Many new and updated translations between CF spec and STASH codes or GRIB2 parameter codes.
- PP/FF loader creates a height coordinate at 1.5m or 10m for certain relevant stash codes.
- Lazy aggregator support for the *standard deviation* aggregator has been added.
- A speed improvement in calculation of *iris.analysis.cartography.area_weights()*.
- Experimental support for unstructured grids has been added with *iris.experimental.ugrid()*. This has been implemented using UGRID.
- *iris.cube.CubeList.extract_overlapping()* supports extraction of cubes over regions where common coordinates overlap, over multiple coordinates.

- Warnings raised due to invalid units in loaded data have been suppressed.
- Experimental low-level read and write access for FieldsFile variants is now supported via `iris.experimental.um.FieldsFileVariant`.
- PP loader will return cubes for all fields prior to a field with a problematic header before raising an exception.
- NetCDF loader skips invalid global attributes, raising a warning rather than raising an exception.
- A warning is now raised rather than an exception when constructing an *AuxCoordFactory* fails.
- Supported *aux coordinate factories* have been extended to include:
- `ocean sigma coordinate`,
- `ocean s coordinate`,
- `ocean s coordinate, generic form 1`, and
- `ocean s coordinate, generic form 2`.
- *iris.cube.Cube.intersection()* now supports taking a points-only intersection. Any bounds on intersected coordinates are ignored but retained.
- The FF loader's known handled grids now includes `Grid 21`.
- A *nearest neighbour* scheme is now provided for *iris.cube.Cube.interpolate()* and *iris.cube.Cube.regrid()*.
- *iris.analysis.cartography.rotate_winds()* supports transformation of wind vectors to a different coordinate system.
- NumPy universal functions can now be applied to cubes using *iris.analysis.maths.apply_ufunc()*.
- Generic functions can be applied to *Cube* instances using *iris.analysis.maths.IFunc*.
- The *iris.analysis.Linear* scheme now supports regridding as well as interpolation. This enables *iris.cube.Cube.regrid()* to perform bilinear regridding, which now replaces the experimental routine "iris.experimental.regrid.regrid_bilinear_rectilinear_src_and_grid".

28.13.2 Bugs Fixed

- Fix in netCDF loader to correctly determine whether the longitude coordinate (including scalar coordinates) is circular.
- *iris.cube.Cube.intersection()* now supports bounds that extend slightly beyond 360 degrees.
- Lateral Boundary Condition (LBC) type FieldFiles are now handled correctly by the FF loader.
- Making a copy of a scalar cube with no data now correctly copies the data array.
- Height coordinates in NAME trajectory output files have been changed to match other NAME output file formats.
- Fixed datatype when loading an `integer_constants` array from a FieldsFile.
- FF/PP loader adds appropriate cell methods for `lbtim.ib = 3` intervals.
- An exception is raised if the units of the latitude and longitude coordinates of the cube passed into *iris.analysis.cartography.area_weights()* are not convertible to radians.
- GRIB1 loader now creates a time coordinate for a time range indicator of 2.
- NetCDF loader now loads units that are empty strings as dimensionless.

v1.8.1 (03 Jun 2015)

- The PP loader now carefully handles floating point errors in date time conversions to hours.
- The handling fill values for lazy data loaded from NetCDF files is altered, such that the _FillValue set in the file is preserved through lazy operations.
- The risk that cube intersections could return incorrect results due to floating point tolerances is reduced.
- The new GRIB2 loading code is altered to enable the loading of various data representation templates; the data value unpacking is handled by the GRIB API.
- Saving cube collections to NetCDF, where multiple similar aux-factories exist within the cubes, is now carefully handled such that extra file variables are created where required in some cases.

28.13.3 Deprecations

- The original GRIB loader has been deprecated and replaced with a new template-based GRIB loader.
- Deprecated default NetCDF save behaviour of assigning the outermost dimension to be unlimited. Switch to the new behaviour with no auto assignment by setting `iris.FUTURE.netcdf_no_unlimited` to True.
- The former experimental method "iris.experimental.regrid.regrid_bilinear_rectilinear_src_and_grid" has been removed, as *`iris.analysis.Linear`* now includes this functionality.

28.13.4 Documentation

- A chapter on *merge and concatenate* has been added to the *user guide*.
- A section on installing Iris using conda has been added to the *install guide*.
- Updates to the chapter on *regridding and interpolation* have been added to the *user guide*.

28.14 v1.7 (04 Jul 2014)

This document explains the changes made to Iris for this release (*View all changes*.)

28.14.1 Features

Showcase: Iris is making use of Biggus

Iris is now making extensive use of Biggus for virtual arrays and lazy array evaluation. In practice this means that analyses of cubes with data bigger than the available system memory are now possible.

Other than the improved functionality the changes are mostly transparent; for example, before the introduction of biggus, MemoryErrors were likely for very large datasets:

```
>>> result = extremely_large_cube.collapsed('time', iris.analyis.MEAN)
MemoryError
```

Now, for supported operations, the evaluation is lazy (i.e. it doesn't take place until the actual data is subsequently requested) and can handle data larger than available system memory:

```
>>> result = extremely_large_cube.collapsed('time', iris.analyis.MEAN)
>>> print(type(result))
<class 'iris.cube.Cube'>
```

Memory is still a limiting factor if ever the data is desired as a NumPy array (e.g. via *cube.data*), but additional methods have been added to the Cube to support querying and subsequently accessing the "lazy" data form (see *has_lazy_data()* and *lazy_data()*).

Showcase: New interpolation and regridding API

New interpolation and regridding interfaces have been added which simplify and extend the existing functionality.

The interfaces are exposed on the cube in the form of the *interpolate()* and *regrid()* methods. Conceptually the signatures of the methods are:

```
interpolated_cube = cube.interpolate(interpolation_points, interpolation_scheme)
```

and:

```
regridded_cube = cube.regrid(target_grid_cube, regridding_scheme)
```

Whilst not all schemes have been migrated to the new interface, *iris.analysis.Linear* defines both linear interpolation and regridding, and *iris.analysis.AreaWeighted* defines an area weighted regridding scheme.

Showcase: Merge and concatenate reporting

Merge reporting is designed as an aid to the merge processes. Should merging a *CubeList* fail, merge reporting means that a descriptive error will be raised that details the differences between the cubes in the *CubeList* that prevented the merge from being successful.

A new *CubeList* method, called *merge_cube()*, has been introduced. Calling it on a *CubeList* will result in a single merged *Cube* being returned or an error message being raised that describes why the merge process failed.

The following example demonstrates the error message that describes a merge failure caused by cubes having differing attributes:

```
>>> cube_list = iris.cube.CubeList((c1, c2))
>>> cube_list.merge_cube()
Traceback (most recent call last):
    ...
    raise iris.exceptions.MergeError(msgs)
iris.exceptions.MergeError: failed to merge into a single cube.
  cube.attributes keys differ: 'foo'
```

The naming of this new method mirrors that of Iris load functions, where one would always expect a *CubeList* from *iris.load()* and a *Cube* from *iris.load_cube()*.

Concatenate reporting is the equivalent process for concatenating a *CubeList*. It is accessed through the method *concatenate_cube()*, which will return a single concatenated cube or produce an error message that describes why the concatenate process failed.

Showcase: Cube broadcasting

When performing cube arithmetic, cubes now follow similar broadcasting rules as NumPy arrays.

However, the additional richness of Iris coordinate meta-data provides an enhanced capability beyond the basic broadcasting behaviour of NumPy.

This means that when performing cube arithmetic, the dimensionality and shape of cubes no longer need to match. For example, if the dimensionality of a cube is reduced by collapsing, then the result can be used to subtract from the original cube to calculate an anomaly:

```
>>> time_mean = original_cube.collapsed('time', iris.analysis.MEAN)
>>> mean_anomaly = original_cube - time_mean
```

Given both broadcasting **and** coordinate meta-data, Iris can now perform arithmetic with cubes that have similar but not identical shape:

```
>>> similar_cube = original_cube.copy()
>>> similar_cube.transpose()
>>> zero_cube = original_cube - similar_cube
```

- Merge reporting that raises a descriptive error if the merge process fails.
- Linear interpolation and regridding now make use of SciPy's RegularGridInterpolator for much faster linear interpolation.
- NAME file loading now handles the "no time averaging" column and translates height/altitude above ground/sea-level columns into appropriate coordinate metadata.
- The NetCDF saver has been extended to allow saving of cubes with hybrid pressure auxiliary factories.
- PP/FF loading supports LBLEV of 9999.
- Extended GRIB1 loading to support data on hybrid pressure levels.
- *iris.coord_categorisation.add_day_of_year()* can be used to add categorised day of year coordinates based on time coordinates with non-Gregorian calendars.
- Support for loading data on reduced grids from GRIB files in raw form without automatically interpolating to a regular grid.
- The coordinate systems *iris.coord_systems.Orthographic* and *iris.coord_systems.VerticalPerspective* (for imagery from geostationary satellites) have been added.
- Extended NetCDF loading to support the "ocean sigma over z" auxiliary coordinate factory.
- Support added for loading CF-NetCDF data with bounds arrays that are missing a vertex dimension.
- *iris.cube.Cube.rolling_window()* can now be used with string-based *iris.coords.AuxCoord* instances.
- Loading of PP and FF files has been optimised through deferring creation of PPField attributes.
- Automatic association of a coordinate's CF formula terms variable with the data variable associated with that coordinate.
- PP loading translates cross-section height into a dimensional auxiliary coordinate.
- String auxiliary coordinates can now be plotted with the Iris plotting wrappers.
- *iris.analysis.geometry.geometry_area_weights()* now allows for the calculation of normalized cell weights.
- Many new translations between the CF spec and STASH codes or GRIB2 parameter codes.

- PP save rules add the data's UM Version to the attributes of the saved file when appropriate.
- NetCDF reference surface variable promotion available through the *iris.FUTURE* mechanism.
- A speed improvement in calculation of *iris.analysis.geometry.geometry_area_weights()*.
- The mdtol keyword was added to area-weighted regridding to allow control of the tolerance for missing data. For a further description of this concept, see *iris.analysis.AreaWeighted*.
- Handling for patching of the CF conventions global attribute via a defined cf_patch_conventions function.
- Deferred GRIB data loading has been introduced for reduced memory consumption when loading GRIB files.
- Concatenate reporting that raises a descriptive error if the concatenation process fails.
- A speed improvement when loading PP or FF data and constraining on STASH code.

28.14.2 Bugs Fixed

- Data containing more than one reference cube for constructing hybrid height coordinates can now be loaded.
- Removed cause of increased margin of error when interpolating.
- Changed floating-point precision used when wrapping points for interpolation.
- Mappables that can be used to generate colorbars are now returned by Iris plotting wrappers.
- NetCDF load ignores over-specified formula terms on bounded dimensionless vertical coordinates.
- Auxiliary coordinate factory loading now correctly interprets formula term variables for "atmosphere hybrid sigma pressure" coordinate data.
- Corrected comparison of NumPy NaN values in cube merge process.
- Fixes for *iris.cube.Cube.intersection()* to correct calculating the intersection of a cube with split bounds, handling of circular coordinates, handling of monotonically descending bounded coordinates and for finding a wrapped two-point result and longitude tolerances.
- A bug affecting *iris.cube.Cube.extract()* and *iris.cube.CubeList.extract()* that led to unexpected behaviour when operating on scalar cubes has been fixed.
- Aggregate_by may now be passed single-value coordinates.
- Making a copy of a *iris.coords.DimCoord* no longer results in the writeable flag on the copied points and bounds arrays being set to True.
- Can now save to PP a cube that has vertical levels but no orography.
- Fix a bug causing surface altitude and surface pressure fields to not appear in cubes loaded with a STASH constraint.
- Fixed support for *iris.fileformats.pp.STASH* objects in STASH constraints.
- A fix to avoid a problem where cube attribute names clash with NetCDF reserved attribute names.
- A fix to allow *iris.cube.CubeList.concatenate()* to deal with descending coordinate order.
- Add missing NetCDF attribute *varname* when constructing a new *iris.coords.AuxCoord*. * The datatype of time arrays converted with *iris.util.unify_time_units()* is now preserved.

v1.7.3 (16 Dec 2014)

- Scalar dimension coordinates can now be concatenated with *iris.cube.CubeList.concatenate()*.
- Arbitrary names can no longer be set for elements of a `iris.fileformats.pp.SplittableInt`.
- Cubes that contain a pseudo-level coordinate can now be saved to PP.
- Fixed a bug in the FieldsFile loader that prevented it always loading all available fields.

v1.7.4 (15 Apr 2015)

- `Coord.guess_bounds()` can now deal with circular coordinates.
- `Coord.nearest_neighbour_index()` can now work with descending bounds.
- Passing *weights* to `Cube.rolling_window()` no longer prevents other keyword arguments from being passed to the aggregator.
- Several minor fixes to allow use of Iris on Windows.
- Made use of the new standard_parallels keyword in Cartopy's LambertConformal projection (Cartopy v0.12). Older versions of Iris will not be able to create LambertConformal coordinate systems with Cartopy >= 0.12.

28.14.3 Incompatible Changes

- Saving a cube with a STASH attribute to NetCDF now produces a variable with an attribute of "um_stash_source" rather than "ukmo__um_stash_source".
- Cubes saved to NetCDF with a coordinate system referencing a spherical ellipsoid now result in the grid mapping variable containing only the "earth_radius" attribute, rather than the "semi_major_axis" and "semi_minor_axis".
- Collapsing a cube over all of its dimensions now results in a scalar cube rather than a 1d cube.

28.14.4 Deprecations

- `iris.util.ensure_array()` has been deprecated.
- Deprecated the `iris.fileformats.pp.reset_load_rules()` and `iris.fileformats.grib.reset_load_rules()` functions.
- Matplotlib is no longer a core Iris dependency.

28.14.5 Documentation

- New sections on *cube broadcasting* and *regridding and interpolation* have been added to the *user guide*.
- An example demonstrating custom log-scale colouring has been added. See *Colouring Anomaly Data With Logarithmic Scaling*.
- An example demonstrating the creation of a custom *iris.analysis.Aggregator* has been added. See *Calculating a Custom Statistic*.
- An example of reprojecting data from 2D auxiliary spatial coordinates (such as that from the ORCA grid) has been added. See *Tri-Polar Grid Projected Plotting*.
- A clarification of the behaviour of *iris.analysis.calculus.differentiate()*.

- A new *"Technical Papers"* section has been added to the documentation along with the addition of a paper providing an *overview of the load process for UM-like fileformats (e.g. PP and Fieldsfile)*.

28.15 v1.6 (26 Jan 2014)

This document explains the changes made to Iris for this release (*View all changes*.)

28.15.1 Features

Showcase Feature - Back to the future ...

The new *iris.FUTURE* global variable is a *iris.Future* instance that controls the run-time behaviour of Iris.

By setting `iris.FUTURE.cell_datetime_objects` to *True*, a *time* reference coordinate will return *datetime-like* objects when invoked with *iris.coords.Coord.cell()* or *iris.coords.Coord.cells()*.

```
>>> from iris.coords import DimCoord
>>> iris.FUTURE.cell_datetime_objects = True
>>> coord = DimCoord([1, 2, 3], "time", units="hours since epoch")
>>> print([str(cell) for cell in coord.cells()])
['1970-01-01 01:00:00', '1970-01-01 02:00:00', '1970-01-01 03:00:00']
```

Note that, either a `datetime.datetime` or `netcdftime.datetime` object instance will be returned, depending on the calendar of the time reference coordinate.

This capability permits the ability to express time constraints more naturally when the cell represents a *datetime-like* object.

```
# Ignore the 1st of January.
iris.Constraint(time=lambda cell: cell.point.month != 1 and cell.point.day != 1)
```

Note that, *iris.Future* also supports a context manager which allows multiple sections of code to execute with different run-time behaviour.

```
>>> print(iris.FUTURE)
Future(cell_datetime_objects=False)
>>> with iris.FUTURE.context(cell_datetime_objects=True):
...     # Code that expects to deal with datetime-like objects.
...     print(iris.FUTURE)
...
Future(cell_datetime_objects=True)
>>> print(iris.FUTURE)
Future(cell_datetime_objects=False)
```

Showcase Feature - Partial date/time ...

The *iris.time.PartialDateTime* class provides the ability to perform comparisons with other *datetime-like* objects such as `datetime.datetime` or `netcdftime.datetime`.

The *year, month, day, hour, minute, second* and *microsecond* attributes of a *iris.time.PartialDateTime* object may be fully or partially specified for any given comparison.

This is particularly useful for time based constraints, whilst enabling the `iris.FUTURE.cell_datetime_objects`, see *here* for further details on this new release feature.

```
from iris.time import PartialDateTime

# Ignore the 1st of January.
iris.Constraint(time=lambda cell: cell != PartialDateTime(month=1, day=1))

# Constrain by a specific year.
iris.Constraint(time=PartialDateTime(year=2013))
```

Also see the User Guide *Constraining on Time* section for further commentary.

- GRIB loading supports latitude/longitude or Gaussian reduced grids for version 1 and version 2.
- *A new utility function to assist with caching*.
- *The RMS aggregator supports weights*.
- *A new experimental function to equalise cube attributes*.
- *Collapsing a cube provides a tolerance level for missing-data*.
- NAME loading supports vertical coordinates.
- UM land/sea mask de-compression for Fieldsfiles and PP files.
- Lateral boundary condition Fieldsfile support.
- Staggered grid support for Fieldsfiles extended to type 6 (Arakawa C grid with v at poles).
- Extend support for Fieldsfiles with grid codes 11, 26, 27, 28 and 29.
- *Promoting a scalar coordinate to new leading cube dimension*.
- Interpreting cell methods from NAME.
- GRIB2 export without forecast_period, enabling NAME to GRIB2.
- Loading height levels from GRIB2.
- *iris.coord_categorisation.add_categorised_coord()* now supports multi-dimensional coordinate categorisation.
- Fieldsfiles and PP support for loading and saving of air potential temperature.
- *iris.experimental.regrid.regrid_weighted_curvilinear_to_rectilinear()* regrids curvilinear point data to a target rectilinear grid using associated area weights.
- Extended capability of the NetCDF saver *iris.fileformats.netcdf.Saver.write()* for fine-tune control of a `netCDF4.Variable`. Also allows multiple dimensions to be nominated as *unlimited*.
- *A new PEAK aggregator providing spline interpolation*.
- A new utility function *iris.util.broadcast_to_shape()*.
- A new utility function *iris.util.as_compatible_shape()*.
- Iris tests can now be run on systems where directory write permissions previously did not allow it. This is achieved by writing to the current working directory in such cases.
- Support for 365 day calendar PP fields.
- Added phenomenon translation between cf and grib2 for wind (from) direction.
- PP files now retain lbfc value on save, derived from the stash attribute.

A New Utility Function to Assist With Caching

To assist with management of caching results to file, the new utility function *iris.util.file_is_newer_than()* may be used to easily determine whether the modification time of a specified cache file is newer than one or more other files.

Typically, the use of caching is a means to circumvent the cost of repeating time consuming processing, or to reap the benefit of fast-loading a pickled cube.

```
# Determine whether to load from the cache or source.
if iris.util.file_is_newer(cache_file, source_file):
    with open(cache_file, "rb") as fh:
        cube = cPickle.load(fh)
else:
    cube = iris.load_cube(source_file)

    # Perhaps perform some intensive processing ...

    # Create the cube cache.
    with open(cache_file, 'wb') as fh:
        cPickle.dump(cube, fh)
```

The RMS Aggregator Supports Weights

The *iris.analysis.RMS* aggregator has been extended to allow the use of weights using the new keyword argument `weights`.

For example, an RMS weighted cube collapse is performed as follows:

```
from iris.analysis import RMS

collapsed_cube = cube.collapsed("height", RMS, weights=weights)
```

Equalise Cube Attributes

To assist with *iris.cube.Cube* merging, the new experimental in-place function *iris.experimental.equalise_cubes.equalise_attributes()* ensures that a sequence of cubes contains a common set of *iris.cube.Cube.attributes*.

This attempts to smooth the merging process by ensuring that all candidate cubes have the same attributes.

Masking a Collapsed Result by Missing-Data Tolerance

The result from collapsing masked cube data may now be completely masked by providing a `mdtol` missing-data tolerance keyword to *iris.cube.Cube.collapsed()*.

This tolerance provides a threshold that will **completely** mask the collapsed result whenever the fraction of data to missing-data is less than or equal to the provided tolerance.

Promote a Scalar Coordinate

The new utility function *iris.util.new_axis()* creates a new cube with a new leading dimension of size unity. If a scalar coordinate is provided, then the scalar coordinate is promoted to be the dimension coordinate for the new leading dimension.

Note that, this function will load the data payload of the cube.

A New PEAK Aggregator Providing Spline Interpolation

The new *iris.analysis.PEAK* aggregator calculates the global peak value from a spline interpolation of the *iris.cube.Cube* data payload along a nominated coordinate axis.

For example, to calculate the peak time:

```
from iris.analysis import PEAK

collapsed_cube = cube.collapsed("time", PEAK)
```

28.15.2 Bugs Fixed

- *iris.cube.Cube.rolling_window()* has been extended to support masked arrays.
- *iris.cube.Cube.collapsed()* now handles string coordinates.
- Default LBUSER(2) to -99 for Fieldsfile and PP saving.
- *iris.util.monotonic()* returns the correct direction.
- File loaders correctly parse filenames containing colons.
- ABF loader now correctly loads the ABF data payload once.
- Support for 1D array `iris.cube.cube.attributes`.
- GRIB bounded level saving fix.
- *iris.analysis.cartography.project()* now associates a coordinate system with the resulting target cube, where applicable.
- *iris.util.array_equal()* now correctly ignores any mask if present, matching the behaviour of `numpy.array_equal()` except with string array support.
- `iris.analysis.interpolate.linear()` now retains a mask in the resulting cube.
- *iris.coords.DimCoord.from_regular()* now correctly returns a coordinate which will always be regular as indicated by *is_regular()*.
- *iris.util.rolling_window()* handling of masked arrays (degenerate masks) fixed.
- Exception no longer raised for any ellipsoid definition in nimrod loading.

28.15.3 Incompatible Changes

- The experimental 'concatenate' function is now a method of a *iris.cube.CubeList*, see *iris.cube.CubeList.concatenate()*. The functionality is unchanged.
- `iris.cube.Cube.extract_by_trajectory()` has been removed. Instead, use *iris.analysis.trajectory.interpolate()*.
- `iris.load_strict()` has been removed. Instead, use *iris.load_cube()* and *iris.load_cubes()*.
- `iris.coords.Coord.cos()` and `iris.coords.Coord.sin()` have been removed.
- `iris.coords.Coord.unit_converted()` has been removed. Instead, make a copy of the coordinate using *iris.coords.Coord.copy()* and then call the *iris.coords.Coord.convert_units()* method of the new coordinate.
- Iteration over a *Cube* has been removed. Instead, use *iris.cube.Cube.slices()*.
- The following `Unit` deprecated methods/properties have been removed.

Removed Property/Method	New Method
`convertible()`	`is_convertible()`
`dimensionless`	`is_dimensionless()`
`no_unit`	`is_no_unit()`
`time_reference`	`is_time_reference()`
`unknown`	`is_unknown()`

- As a result of deprecating `iris.cube.Cube.add_history()` and removing the automatic appending of history by operations such as cube arithmetic, collapsing, and aggregating, the signatures of a number of functions within *iris.analysis.maths* have been modified along with that of *iris.analysis.Aggregator* and *iris.analysis.WeightedAggregator*.
- The experimental ABF and ABL functionality has now been promoted to core functionality in *iris.fileformats.abf*.
- The following *iris.coord_categorisation* deprecated functions have been removed.

Removed Function	New Function
`add_custom_season()`	*add_season()*
`add_custom_season_number()`	*add_season_number()*
`add_custom_season_year()`	*add_season_year()*
`add_custom_season_membership()`	*add_season_membership()*
`add_month_shortname()`	*add_month()*
`add_weekday_shortname()`	*add_weekday()*
`add_season_month_initials()`	*add_season()*

- When a cube is loaded from PP or GRIB and it has both time and forecast period coordinates, and the time coordinate has bounds, the forecast period coordinate will now also have bounds. These bounds will be aligned with the bounds of the time coordinate taking into account the forecast reference time. Also, the forecast period point will now be aligned with the time point.

28.15.4 Deprecations

- `iris.cube.Cube.add_history()` has been deprecated in favour of users modifying/creating the history metadata directly. This is because the automatic behaviour did not deliver a sufficiently complete, auditable history and often prevented the merging of cubes.
- `iris.util.broadcast_weights()` has been deprecated and replaced by the new utility function *iris.util.broadcast_to_shape()*.
- Callback mechanism *iris.run_callback* has had its deprecation of return values revoked. The callback can now return cube instances as well as inplace changes to the cube.

28.15.5 New Contributors

Congratulations and thank you to felicityguest, jkettleb, kwilliams-mo and shoyer who all made their first contribution to Iris!

28.16 v1.5 (13 Sep 2013)

This document explains the changes made to Iris for this release (*View all changes*.)

28.16.1 Features

- Scatter plots can now be produced using *iris.plot.scatter()* and *iris.quickplot.scatter()*.
- The functions *iris.plot.plot()* and *iris.quickplot.plot()* now take up to two arguments, which may be cubes or coordinates, allowing the user to have full control over what is plotted on each axis. The coords keyword argument is now deprecated for these functions. This now also gives extended 1D plotting capability.

```
# plot a 1d cube against a given 1d coordinate, with the cube
# values on the x-axis and the coordinate on the y-axis
iris.plot.plot(cube, coord)
```

- *iris.analysis.SUM* is now a weighted aggregator, allowing it to take a weights keyword argument.
- GRIB2 translations added for standard_name 'soil_temperature'.
- *iris.cube.Cube.slices()* can now handle passing dimension index as well as the currently supported types (string, coordinate), in order to slice in cases where there is no coordinate associated with a dimension (a mix of types is also supported).

```
# Get cube slices corresponding to the dimension associated with longitude
# and the first dimension from a multi-dimensional cube.
for sub_cube in cube.slices(['longitude', 0]):
    print(sub_cube)
```

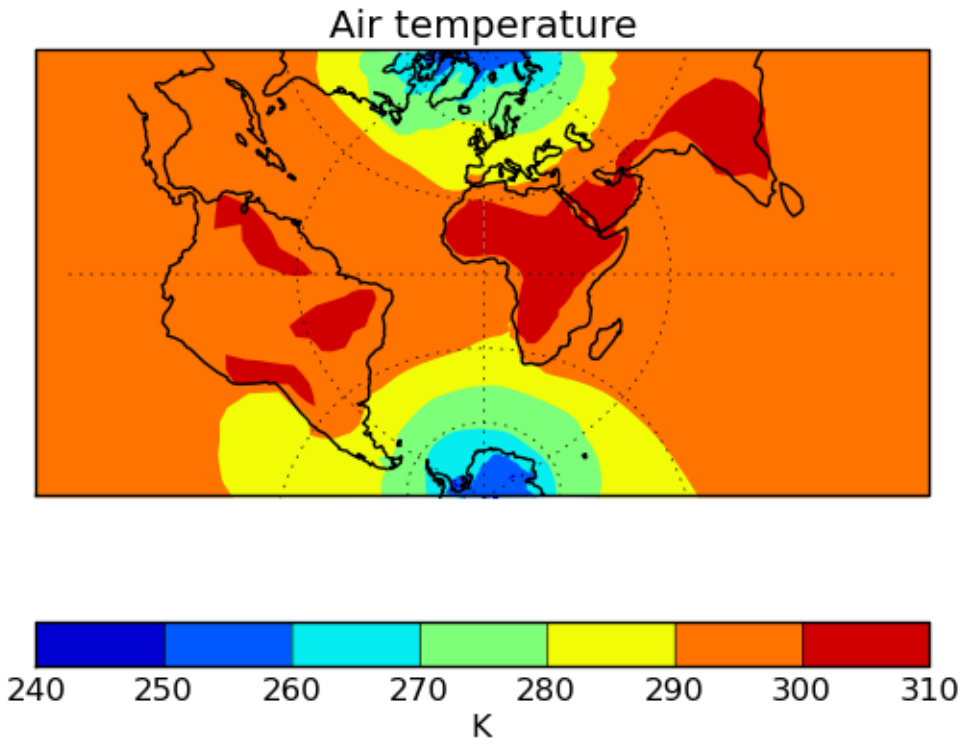
- *iris.experimental.animate* now provides experimental animation support.

```
# Create an iterable of cubes (generator, lists etc.)
cube_iter = cubes.slices(('grid_longitude', 'grid_latitude'))
ani = animate(cube_iter, qplt.contourf)
plt.show()
```

- Support for UM ancillary files truncated with the UM utility ieee

- Complete support for Transverse Mercator with saving to NetCDF also.

```
import cartopy.crs as ccrs
import iris
import iris.quickplot as qplt
import matplotlib.pyplot as plt


fname = iris.sample_data_path('air_temp.pp')
temperature = iris.load_cube(fname)

plt.axes(projection=ccrs.TransverseMercator())
qplt.contourf(temperature)
plt.gca().coastlines()
plt.gca().gridlines()
plt.show()
```



- Support for loading NAME files (gridded and trajectory data).
- Multi-dimensional coordinate support added for *iris.analysis.cartography.cosine_latitude_weights()*
- Added limited packaged GRIB support (bulletin headers).
- In-place keyword added to *iris.analysis.maths.divide()* and *iris.analysis.maths.multiply()*.
- Performance gains for PP loading of the order of 40%.
- *iris.quickplot* now has a `show()` function to provide convenient access to matplotlib.pyplot.show().

- *iris.coords.DimCoord.from_regular()* now implemented which creates a *DimCoord* with regularly spaced points, and optionally bounds.
- Iris can now cope with a missing bounds variable from NetCDF files.
- Added support for bool array indexing on a cube.

```
fname = iris.sample_data_path('air_temp.pp')
temperature = iris.load_cube(fname)
temperature[temperature.coord('latitude').points > 0]

# The constraints mechanism is still the preferred means to do such a query.
temperature.extract(iris.Constraint(latitude=lambda v: v>0)))
```

- Added support for loading fields defined on regular Gaussian grids from GRIB files.
- `iris.analysis.interpolate.extract_nearest_neighbour()` now works without needing to load the data (especially relevant to large datasets).
- When using plotting routines from *iris.plot* or *iris.quickplot*, the direction of vertical axes will be reversed if the corresponding coordinate has a "positive" attribute set to "down".

  see: *Oceanographic Profiles and T-S Diagrams*
- New PP stashcode translations added including 'dewpoint' and 'relative_humidity'.
- Added implied heights for several common PP STASH codes.
- GeoTIFF export capability enhanced for supporting various data types, coord systems and mapping 0 to 360 longitudes to the -180 to 180 range.

28.16.2 Bugs Fixed

- NetCDF error handling on save has been extended to capture file path and permission errors.
- Shape of the Earth scale factors are now correctly interpreted by the GRIB loader. They were previously used as a multiplier for the given value but should have been used as a decimal shift.
- OSGB definition corrected.
- Transverse Mercator on load now accepts the following interchangeably due to inconsistencies in CF documentation:
  - +scale_factor_at_central_meridian <-> scale_factor_at_projection_origin
  - +longitude_of_central_meridian <-> longitude_of_projection_origin (+recommended encoding)
- Ellipse description now maintained when converting GeogCS to cartopy.
- GeoTIFF export bug fixes.
- Polar axis now set to the North Pole, when a cube with no coordinate system is saved to the PP file-format.
- *iris.coords.DimCoord.from_coord()* and *iris.coords.AuxCoord.from_coord()* now correctly returns a copy of the source coordinate's coordinate system.
- Units part of the axis label is now omitted when the coordinate it represents is given as a time reference (*iris.quickplot*).
- CF dimension coordinate is now maintained in the resulting cube when a cube with CF dimension coordinate is being aggregated over.
- Units for Lambert conformal and polar stereographic coordinates now defined as meters.

- Various fieldsfile load bugs including failing to read the coordinates from the file have been fixed.
- Coding of maximum and minimum time-stats in GRIB2 saving has been fixed.
- Example code in section 4.1 of the user guide updated so it uses a sample data file that exists.
- Zorder of contour lines drawn by *contourf()* has been changed to address issue of objects appearing in-between line and filled contours.
- Coord comparisons now function correctly when comparing to numpy scalars.
- Cube loading constraints and *iris.cube.Cube.extract()* correctly implement cell equality methods.

28.16.3 Deprecations

- The coords keyword argument for *iris.plot.plot()* and *iris.quickplot.plot()* has been deprecated due to the new API which accepts multiple cubes or coordinates.
- `iris.fileformats.pp.PPField.regular_points()` and `iris.fileformats.pp.PPField.regular_bounds()` have now been deprecated in favour of a new factory method *iris.coords.DimCoord.from_regular()*.
- `iris.fileformats.pp.add_load_rules()` and `iris.fileformats.grib.add_load_rules()` are now deprecated.

28.17 v1.4 (14 Jun 2013)

This document explains the changes made to Iris for this release (*View all changes*.)

28.17.1 Features

- Multiple cubes can now be exported to a NetCDF file.
- Correct nearest-neighbour calculation with circular coords.
- *Experimental regridding enhancements*.
- *Iris-Pandas interoperability*.
- NIMROD level type 12 (levels below ground) can now be loaded.
- *Load cubes from the internet via OPeNDAP*.
- *GeoTiff export (experimental)*.
- *Cube merge update*.
- *Unambiguous season year naming*.
- NIMROD files with multiple fields and period of interest can now be loaded.
- Missing values are now handled when loading GRIB messages.
- PP export rule to calculate forecast period.
- *aggregated_by()* now maintains array masking.
- IEEE 32bit fieldsfiles can now be loaded.
- NetCDF transverse mercator and climatology data can now be loaded.
- Polar stereographic GRIB data can now be loaded.

- *Cubes with no vertical coord can now be exported to GRIB*.
- *Simplified resource configuration*.
- *Extended GRIB parameter translation*.
- Added an optimisation for single-valued coordinate constraints.
- *One dimensional linear interpolation fix*.
- *Fix for iris.analysis.calculus.differentiate*.
- Fixed pickling of cubes with 2D aux coords from NetCDF.
- Fixed bug which ignored the "coords" keyword for certain plots.
- Use the latest release of Cartopy, v0.8.0.

Experimental Regridding Enhancements

Bilinear, area-weighted and area-conservative regridding functions are now available in *iris.experimental*. These functions support masked data and handle derived coordinates such as hybrid height. The final API is still in development.

In the meantime:

Bilinear Rectilinear Regridding

`regrid_bilinear_rectilinear_src_and_grid()` can be used to regrid a cube onto a horizontal grid defined in a differentiate coordinate system. The data values are calculated using bilinear interpolation.

For example:

```
from iris.experimental.regrid import regrid_bilinear_rectilinear_src_and_grid
regridded_cube = regrid_bilinear_rectilinear_src_and_grid(source_cube, target_grid_cube)
```

Area-Weighted Regridding

regrid_area_weighted_rectilinear_src_and_grid() can be used to regrid a cube such that the data values of the resulting cube are calculated using the area-weighted mean.

For example:

```
from iris.experimental.regrid import regrid_area_weighted_rectilinear_src_and_grid as␣
↪regrid_area_weighted
regridded_cube = regrid_area_weighted(source_cube, target_grid_cube)
```

Area-Conservative Regridding

regrid_conservative_via_esmpy() can be used for area-conservative regridding between geographical coordinate systems. This uses the ESMF library functions, via the ESMPy interface.

For example:

```
from iris.experimental.regrid_conservative import regrid_conservative_via_esmpy
regridded_cube = regrid_conservative_via_esmpy(source_cube, target_grid_cube)
```

Iris-Pandas Interoperability

Conversion to and from Pandas Series and DataFrames is now available. See *iris.pandas* for more details.

Load Cubes From the Internet via OPeNDAP

Cubes can now be loaded directly from the internet, via OPeNDAP.

For example:

```
cubes = iris.load("http://geoport.whoi.edu/thredds/dodsC/bathy/gom15")
```

GeoTiff Export

With this experimental feature, two dimensional cubes can now be exported to GeoTiff files.

For example:

```
from iris.experimental.raster import export_geotiff
export_geotiff(cube, filename)
```

Note: This is a raw data export only and does not save Iris plots.

Cube Merge Update

Cube merging now favours numerical coordinates over string coordinates to describe a dimension, and *DimCoord* over *AuxCoord*. These modifications prevent the error: *"No functional relationship between separable and inseparable candidate dimensions"*.

Unambiguous Season Year Naming

The default names of categorisation coordinates are now less ambiguous. For example, *add_month_number()* and *add_month_fullname()* now create "month_number" and "month_fullname" coordinates.

Cubes With no Vertical Coord can now be Exported to GRIB

Iris can now export cubes with no vertical coord to GRIB. The solution is still under discussion: See https://github.com/SciTools/iris/issues/519.

Simplified Resource Configuration

A new configuration variable called `iris.config.TEST_DATA_DIR` has been added, replacing the previous combination of `iris.config.MASTER_DATA_REPOSITORY` and `iris.config.DATA_REPOSITORY`. This constant should be the path to a directory containing the test data required by the unit tests. It can be set by adding a `test_data_dir` entry to the `Resources` section of `site.cfg`. See *iris.config* for more details.

Extended GRIB Parameter Translation

- More GRIB2 params are recognised on input.
- Now translates some codes on GRIB2 output.
- Some GRIB2 params may load with a different standard_name.

One dimensional Linear Interpolation Fix

`linear()` can now extrapolate from a single point assuming a gradient of zero. This prevents an issue when loading cross sections with a hybrid height coordinate, on a staggered grid and only a single orography field.

Fix for iris.analysis.calculus.differentiate

A bug in *differentiate()* that had the potential to cause the loss of coordinate metadata when calculating the curl or the derivative of a cube has been fixed.

28.17.2 Incompatible Changes

- As part of simplifying the mechanism for accessing test data, `iris.io.select_data_path()`, `iris.config.DATA_REPOSITORY`, `iris.config.MASTER_DATA_REPOSITORY` and `iris.config.RESOURCE_DIR` have been removed.

28.17.3 Deprecations

- The *add_custom_season_** functions from *coord_categorisation* have been deprecated in favour of adding their functionality to the *add_season_** functions

28.18 v1.3 (27 Mar 2013)

This document explains the changes made to Iris for this release (*View all changes*.)

28.18.1 Features

- Experimental support for *loading ABF/ABL files*.
- Support in `iris.analysis.interpolate.linear()` for longitude ranges other than [-180, 180].
- Support for *customised CF profiles* on export to netCDF.
- The documentation now includes guidance on *how to cite Iris*.
- The ability to calculate the exponential of a Cube, via *`iris.analysis.maths.exp()`*.
- Experimental support for *concatenating Cubes* along existing dimensions via `iris.experimental.concatenate.concatenate()`.

Loading ABF/ABL Files

Support for the ABF and ABL file formats (as defined by the climate and vegetation research group of Boston University), is currently provided under the "experimental" system. As such, ABF/ABL file detection is not automatically enabled.

To enable ABF/ABL file detection, simply import the `iris.experimental.fileformats.abf` module before attempting to load an ABF/ABL file.

For example:

```
import iris.experimental.fileformats.abf
cube = iris.load_cube('/path/to/my/data.abf')
```

Customised CF Profiles

Iris now provides hooks in the CF-netCDF export process to allow user-defined routines to check and/or modify the representation in the netCDF file.

The following keys within the `iris.site_configuration` dictionary have been **reserved** as hooks to *external* user-defined CF profile functions:

- `cf_profile` ingests a *`iris.cube.Cube`* for analysis and returns a profile result
- `cf_patch` modifies the CF-netCDF file associated with export of the *`iris.cube.Cube`*

The `iris.site_configuration` dictionary should be configured via the `iris/site_config.py` file.

For further implementation details see `iris/fileformats/netcdf.py`.

Cube Concatenation

Iris now provides initial support for concatenating Cubes along one or more existing dimensions. Currently this will force the data to be loaded for all the source Cubes, but future work will remove this restriction.

For example, if one began with a collection of Cubes, each containing data for a different range of times:

```
>>> print(cubes)
0: air_temperature                      (time: 30; latitude: 145; longitude: 192)
1: air_temperature                      (time: 30; latitude: 145; longitude: 192)
2: air_temperature                      (time: 30; latitude: 145; longitude: 192)
```

One could use `iris.experimental.concatenate.concatenate()` to combine these into a single Cube as follows:

```
>>> new_cubes = iris.experimental.concatenate.concatenate(cubes)
>>> print(new_cubes)
0: air_temperature                      (time: 90; latitude: 145; longitude: 192)
```

Note: As this is an experimental feature, your feedback is especially welcome.

28.18.2 Bugs Fixed

- Printing a Cube now supports Unicode attribute values.
- PP export now sets LBMIN correctly.
- Converting between reference times now works correctly for units with non-Gregorian calendars.
- Slicing a *CubeList* now returns a *CubeList* instead of a normal list.

28.18.3 Deprecations

- The boolean methods/properties on the `Unit` class have been updated to *is_...()* methods, in line with the project's naming conventions.

Deprecated Property/Method	New Method
`convertible()`	`is_convertible()`
`dimensionless`	`is_dimensionless()`
`no_unit`	`is_no_unit()`
`time_reference`	`is_time_reference()`
`unknown`	`is_unknown()`

28.19 v1.2 (28 Feb 2013)

This document explains the changes made to Iris for this release (*View all changes*.)

28.19.1 Features

- *iris.cube.Cube.convert_units()* and *iris.coords.Coord.convert_units()* have been added. This is aimed at simplifying the conversion of a cube or coordinate from one unit to another. For example, to convert a cube in kelvin to celsius, one can now call cube.convert_units('celsius'). The operation is in-place and if the units are not convertible an exception will be raised.
- *iris.cube.Cube.var_name*, *iris.coords.Coord.var_name* and *iris.aux_factory.AuxCoordFactory.var_name* attributes have been added. This attribute represents the CF variable name of the object. It is populated when loading from CF-netCDF files and is used when writing to CF-netCDF. A var_name keyword argument has also been added to the *iris.cube.Cube.coord()*, *iris.cube.Cube.coords()* and *iris.cube.Cube.aux_factory()* methods.
- *iris.coords.Coord.is_compatible()* has been added. This method is used to determine whether two coordinates are sufficiently alike to allow operations such as *iris.coords.Coord.intersect()* and `iris.analysis.interpolate.regrid()` to take place. A corresponding method for cubes, *iris.cube.Cube.is_compatible()*, has also been added.
- Printing a *Cube* is now more user friendly with regards to dates and time. All *time* and *forecast_reference_time* scalar coordinates now display human readable date/time information.
- The units of a *Cube* are now shown when it is printed.
- The area weights calculated by *iris.analysis.cartography.area_weights()* may now be normalised relative to the total grid area.
- Weights may now be passed to *iris.cube.Cube.rolling_window()* aggregations, thus allowing arbitrary digital filters to be applied to a *Cube*.

28.19.2 Bugs Fixed

- The GRIB hindcast interpretation of negative forecast times can be enabled via the `iris.fileformats.grib.hindcast_workaround` flag.
- The NIMROD file loader has been extended to cope with orography vertical coordinates.

28.19.3 Incompatible Changes

- The deprecated `iris.cube.Cube.unit` and `iris.coords.Coord.unit` attributes have been removed.

28.19.4 Deprecations

- The `iris.coords.Coord.unit_converted()` method has been deprecated. Users should make a copy of the coordinate using *iris.coords.Coord.copy()* and then call the *iris.coords.Coord.convert_units()* method of the new coordinate.
- With the addition of the var_name attribute the signatures of DimCoord and AuxCoord have changed. This should have no impact if you are providing parameters as keyword arguments, but it may cause issues if you are relying on the position/order of the arguments.
- Iteration over a *Cube* has been deprecated. Instead, users should use *iris.cube.Cube.slices()*.

28.20 v1.1 (03 Jan 2013)

This document explains the changes made to Iris for this release (*View all changes*.)

28.20.1 Features

With the release of Iris 1.1, we are introducing support for Mac OS X. Version 1.1 also sees the first batch of performance enhancements, with some notable improvements to netCDF/PP import.

- Support for Mac OS X.
- GRIB1 import now supports time units of "3 hours".
- Fieldsfile import now supports unpacked and "CRAY" 32-bit packed data in 64-bit Fieldsfiles.
- PP file import now supports "CRAY" 32-bit packed data.
- Various performance improvements, particularly for netCDF import, PP import, and constraints.
- GRIB2 export now supports level types of altitude and height (codes 102 and 103).
- iris.analysis.cartography.area_weights now supports non-standard dimension orders.
- PP file import now adds the "forecast_reference_time" for fields where LBTIM is 11, 12, 13, 31, or 32.
- PP file import now supports LBTIM values of 1, 2, and 3.
- Fieldsfile import now has some support for ancillary files.
- Coordinate categorisation functions added for day-of-year and user-defined seasons.
- GRIB2 import now has partial support for probability data defined with product template 4.9.

Coordinate Categorisation

An *add_day_of_year()* categorisation function has been added to the existing suite in *iris.coord_categorisation*.

Custom Seasons

The conventional seasonal categorisation functions have been complemented by two groups of functions which handle user-defined, custom seasons.

The first group of functions is:

- `iris.coord_categorisation.add_custom_season()`
- `iris.coord_categorisation.add_custom_season_number()`
- `iris.coord_categorisation.add_custom_season_year()`

These functions mimic their non-custom versions, but with the addition of a `seasons` parameter which is used to define the custom seasons. These seasons are defined by concatenating the single letter abbreviations of the relevant, consecutive months.

For example, to categorise a Cube based on "winter" and "summer" months, one might do:

```
>>> seasons = ['mamjja', 'sondjf']
>>> iris.coord_categorisation.add_custom_season(cube, 'time', seasons)
>>> print(cube.coord('season').points)
['ondjfm' 'ondjfm' 'mamjja' 'mamjja' 'mamjja' 'mamjja' 'mamjja' 'mamjja'
 'ondjfm' 'ondjfm' 'ondjfm' 'ondjfm']
```

The other custom season function is:

- `iris.coord_categorisation.add_custom_season_membership()`.

This function adds a coordinate containing True/False values determined by membership of a single custom season.

28.20.2 Bugs Fixed

- PP export no longer attempts to set/overwrite the STASH code based on the standard_name.
- Cell comparisons now work consistently, which fixes a bug where bounded_cell > point_cell compares the point to the bounds but, point_cell < bounded_cell compares the points.
- Fieldsfile import now correctly recognises pre v3.1 and post v5.2 versions, which fixes a bug where the two were interchanged.
- iris.analysis.trajectory.interpolate now handles hybrid-height.

28.21 v1.0 (17 Oct 2012)

This document explains the changes made to Iris for this release (*View all changes*.)

With the release of Iris 1.0, we have broadly completed the transition to the CF data model, and established a stable foundation for future work. Following this release we plan to deliver significant performance improvements and additional features.

28.21.1 The Role of 1.x

The 1.x series of releases is intended to provide a relatively stable, backwards-compatible platform based on the CF-netCDF data model, upon which long-lived services can be built.

Iris 1.0 targets the data model implicit in CF-netCDF 1.5. This will be extended to cover the new features of CF-netCDF 1.6 (e.g. discrete sampling geometries) and any subsequent versions which maintain backwards compatibility. Similarly, as the efforts of the CF community to formalise their data model reach maturity, they will be included in Iris where significant backwards-compatibility can be maintained.

28.21.2 Features

A summary of the main features added with version 1.0:

- Hybrid-pressure vertical coordinates, and the ability to load from GRIB.
- Initial support for CF-style coordinate systems.
- Use of Cartopy for mapping in matplotlib.
- Load data from NIMROD files.
- Availability of Cynthia Brewer colour palettes.
- Add a citation to a plot.
- Ensures netCDF files are properly closed.
- The ability to bypass merging when loading data.
- Save netCDF files with an unlimited dimension.
- A more explicit set of load functions, which also allow the automatic cube merging to be bypassed as a last resort.
- The ability to project a cube with a lat-lon or rotated lat-lon coordinate system into a range of map projections e.g. Polar Stereographic.
- Cube summaries are now more readable when the scalar coordinates contain bounds.

CF-NetCDF Coordinate Systems

The coordinate systems in Iris are now defined by the CF-netCDF grid mappings. As of Iris 1.0 a subset of the CF-netCDF coordinate systems are supported, but this will be expanded in subsequent versions. Adding this code is a relatively simple, incremental process - it would make a good task to tackle for users interested in getting involved in contributing to the project.

The coordinate systems available in Iris 1.0 and their corresponding Iris classes are:

CF Name	Iris Class
Latitude-longitude	*GeogCS*
Rotated pole	*RotatedGeogCS*
Transverse Mercator	*TransverseMercator*

For convenience, Iris also includes the *OSGB* class which provides a simple way to create the transverse Mercator coordinate system used by the British Ordnance Survey.

Using Cartopy for Mapping in Matplotlib

The underlying map drawing package has now been updated to use Cartopy. Cartopy provides a highly flexible set of mapping tools, with a consistent, intuitive interface. As yet it doesn't have feature-parity with basemap, but its goal is to make maps "just work", making it the perfect complement to Iris.

The `iris.plot.map_setup` function has now been replaced with a cleaner interface:

- To draw a cube on its native map project, one can simply draw the cube directly:

```
import iris.plot as iplt
import matplotlib.pyplot as plt

iplt.contourf(cube)
plt.gca().coastlines()
plt.show()
```

- To draw a cube on the native map and extents of another, one can use the *iris.plot.default_projection()* and *iris.plot.default_projection_extent()* functions:

```
import iris.plot as iplt
import matplotlib.pyplot as plt

cube1_projection = iplt.default_projection(cube1)
cube1_extent = iplt.default_projection_extent(cube1)

ax = plt.axes(projection=cube1_projection)
ax.set_extent(cube1_extent, cube1_projection)
iplt.contourf(cube2)
ax.coastlines()
plt.show()
```

Note: The `iris.plot.gcm` function to get the current map is now redundant; instead the current map *is* the current matplotlib axes, and `matplotlib.pyplot.gca()` should be used instead.

For more examples of what can be done with Cartopy, see the Iris gallery and Cartopy's documentation.

Hybrid-Pressure

With the introduction of the *HybridPressureFactory* class, it is now possible to represent data expressed on a hybrid-pressure vertical coordinate. A hybrid-pressure factory is created with references to the coordinates which provide the components of the hybrid coordinate ("ap" and "b") and the surface pressure. In return, it provides a virtual "pressure" coordinate whose values are derived from the given components.

This facility is utilised by the GRIB2 loader to automatically provide the derived "pressure" coordinate for certain data[1] from the ECMWF.

[1] Where the level type is either 105 or 119, and where the surface pressure has an ECMWF paramId of 152.

NetCDF

When saving a Cube to a netCDF file, Iris will now define the outermost dimension as an unlimited/record dimension. In combination with the *iris.cube.Cube.transpose()* method, this allows any dimension to take the role of the unlimited/record dimension.

For example, a Cube with the structure:

```
<iris 'Cube' of air_potential_temperature (time: 6; model_level_number: 70; grid_
↪latitude: 100; grid_longitude: 100)>
```

would result in a netCDF file whose CDL definition would include:

```
dimensions:
        time = UNLIMITED ; // (6 currently)
        model_level_number = 70 ;
        grid_latitude = 100 ;
        grid_longitude = 100 ;
```

Also, Iris will now ensure that netCDF files are properly closed when they are no longer in use. Previously this could cause problems when dealing with large numbers of netCDF files, or in long running processes.

Brewer Colour Palettes

Iris includes a selection of carefully designed colour palettes produced by Cynthia Brewer. The *iris.palette* module registers the Brewer colour palettes with matplotlib, so they are explicitly selectable via the `matplotlib.pyplot.set_cmap()` function. For example:

```
import iris.palette
import matplotlib.pyplot as plt
import numpy as np
plt.contourf(np.random.randn(10, 10))
plt.set_cmap('brewer_RdBu_11')
plt.show()
```

Citations

Citations can easily be added to a plot using the *iris.plot.citation()* function. The recommended text for the Cynthia Brewer citation is provided by `iris.plot.BREWER_CITE`.

To include a reference in a journal article or report please refer to section 5 in the citation guidance provided by Cynthia Brewer.

Metadata Attributes

Iris now stores "source" and "history" metadata in Cube attributes. For example:

```
>>> print(iris.tests.stock.global_pp())
air_temperature                     (latitude: 73; longitude: 96)
     ...
     Attributes:
          ...
```

(continues on next page)

(continued from previous page)

```
        source: Data from Met Office Unified Model
    ...
```

Where previously it would have appeared as:

```
air_temperature                     (latitude: 73; longitude: 96)
    ...
    Scalar coordinates:
        ...
        source: Data from Met Office Unified Model
    ...
```

Note: This change breaks backwards compatibility with Iris 0.9. But if it is desirable to have the "source" metadata expressed as a coordinate then it can be done with the following pattern:

```
src = cube.attributes.pop('source')
src_coord = iris.coords.AuxCoord(src, long_name='source')
cube.add_aux_coord(src_coord)
```

New Loading Functions

The main functions for loading cubes are now:

- *iris.load()*
- *iris.load_cube()*
- *iris.load_cubes()*

These provide convenient cube loading suitable for both interactive (*iris.load()*) and scripted (*iris.load_cube()*, *iris.load_cubes()*) usage.

In addition, *iris.load_raw()* has been provided as a last resort for situations where the automatic cube merging is not appropriate. However, if you find you need to use this function we would encourage you to contact the Iris developers so we can see if a fix can be made to the cube merge algorithm.

The `iris.load_strict()` function has been deprecated. Code should now use the *iris.load_cube()* and *iris.load_cubes()* functions instead.

Cube Projection

Iris now has the ability to project a cube into a number of map projections. This functionality is provided by *iris.analysis.cartography.project()*. For example:

```
import iris
import cartopy.crs as ccrs
import matplotlib.pyplot as plt

# Load data
cube = iris.load_cube(iris.sample_data_path('air_temp.pp'))

# Transform cube to target projection
```

(continues on next page)

(continued from previous page)

```
target_proj = ccrs.RotatedPole(pole_longitude=177.5,
                               pole_latitude=37.5)
new_cube, extent = iris.analysis.cartography.project(cube, target_proj)

# Plot
plt.axes(projection=target_proj)
plt.pcolor(new_cube.coord('projection_x_coordinate').points,
           new_cube.coord('projection_y_coordinate').points,
           new_cube.data)
plt.gca().coastlines()
plt.show()
```

This function is intended to be used in cases where the cube's coordinates prevent one from directly visualising the data, e.g. when the longitude and latitude are two dimensional and do not make up a regular grid. The function uses a nearest neighbour approach rather than any form of linear/non-linear interpolation to determine the data value of each cell in the resulting cube. Consequently it may have an adverse effect on the statistics of the data e.g. the mean and standard deviation will not be preserved. This function currently assumes global data and will if necessary extrapolate beyond the geographical extent of the source cube.

28.21.3 Incompatible Changes

- The "source" and "history" metadata are now represented as Cube attributes, where previously they used coordinates.
- *iris.cube.Cube.coord_dims()* now returns a tuple instead of a list.
- The `iris.plot.gcm` and `iris.plot.map_setup` functions are now removed. See *Using Cartopy for Mapping in Matplotlib* for further details.

28.21.4 Deprecations

- The methods `iris.coords.Coord.cos()` and `iris.coords.Coord.sin()` have been deprecated.
- The `iris.load_strict()` function has been deprecated. Code should now use the *iris.load_cube()* and *iris.load_cubes()* functions instead.

CHAPTER

TWENTYNINE

IRIS TECHNICAL PAPERS

Extra information on specific technical issues.

29.1 Iris Handling of PP and Fieldsfiles

This document provides a basic account of how PP and Fieldsfiles data is represented within Iris. It describes how Iris represents data from the Met Office Unified Model (UM), in terms of the metadata elements found in PP and Fieldsfile formats.

For simplicity, we shall describe this mostly in terms of *loading of PP data into Iris* (i.e. into cubes). However most of the details are identical for Fieldsfiles, and are relevant to saving in these formats as well as loading.

Notes:

1. Iris treats Fieldsfile data almost exactly as if it were PP – i.e. it treats each field's lookup table entry like a PP header.
2. The Iris data model is based on NetCDF CF conventions, so most of this can also be seen as a metadata translation between PP and CF terms, but it is easier to discuss in terms of Iris elements.

For details of Iris terms (cubes, coordinates, attributes), refer to *Iris data structures*.

For details of CF conventions, see http://cfconventions.org/.

29.1.1 Overview of Loading Process

The basics of Iris loading are explained at *Loading Iris Cubes*. Loading as it specifically applies to PP and Fieldsfile data can be summarised as follows:

1. Input fields are first loaded from the given sources, using `iris.fileformats.pp.load()`. This returns an iterator, which provides a 'stream' of `PPField` input field objects. Each PPfield object represents a single source field:
   - PP header elements are provided as named object attributes (e.g. `lbproc`).
   - Some extra, calculated "convenience" properties are also provided (e.g. `t1` and `t2` time values).
   - There is a `iris.fileformats.pp.PPField.data` attribute, but the field data is not actually loaded unless/until this is accessed, for greater speed and space efficiency.
2. Each input field is translated into a two-dimensional Iris cube (with dimensions of latitude and longitude). These are the 'raw' cubes, as returned by `iris.load_raw()`. Within these:
   - There are 2 horizontal dimension coordinates containing the latitude and longitude values for the field.

- Certain other header elements are interpreted as 'coordinate'-type values applying to the input fields, and stored as auxiliary 'scalar' (i.e. 1-D) coordinates. These include all header elements defining vertical and time coordinate values, and also more specialised factors such as ensemble number and pseudo-level.
- Other metadata is encoded on the cube in a variety of other forms, such as the cube 'name' and 'units' properties, attribute values and cell methods.

3. Lastly, Iris attempts to merge the raw cubes into higher-dimensional ones (using *merge()*): This combines raw cubes with different values of a scalar coordinate to produce a higher-dimensional cube with the values contained in a new vector coordinate. Where possible, the new vector coordinate is also a *dimension* coordinate, describing the new dimension. Apart from the original 2 horizontal dimensions, all cube dimensions and dimension coordinates arise in this way – for example, 'time', 'height', 'forecast_period', 'realization'.

Note: This document covers the essential features of the UM data loading process. The complete details are implemented as follows:

- The conversion of fields to raw cubes is performed by the function `iris.fileformats.pp_rules.convert()`, which is called from *iris.fileformats.pp.load_cubes()* during loading.
- The corresponding save functionality for PP output is implemented by the *iris.fileformats.pp.save()* function. The relevant 'save rules' are defined in a text file ("lib/iris/etc/pp_save_rules.txt"), in a form defined by the *iris.fileformats.rules* module.

The rest of this document describes various independent sections of related metadata items.

29.1.2 Horizontal Grid

UM Field elements LBCODE, BPLAT, BPLON, BZX, BZY, BDX, BDY, X, Y, X_LOWER_BOUNDS, Y_LOWER_BOUNDS

Cube components (unrotated) : coordinates `longitude`, `latitude`

(rotated pole) : coordinates `grid_latitude`, `grid_longitude`

Details

At present, only latitude-longitude projections are supported (both normal and rotated). In these cases, LBCODE is typically 1 or 101 (though, in fact, cross-sections with latitude and longitude axes are also supported).

For an ordinary latitude-longitude grid, the cubes have coordinates called 'longitude' and 'latitude':

- These are mapped to the appropriate data dimensions.
- They have units of 'degrees'.
- They have a coordinate system of type *iris.coord_systems.GeogCS*.
- The coordinate points are normally set to the regular sequence `ZDX/Y + BDX/Y * (1 .. LBNPT/LBROW)` (*except*, if BDX/BDY is zero, the values are taken from the extra data vector X/Y, if present).
- If X/Y_LOWER_BOUNDS extra data is available, this appears as bounds values of the horizontal coordinates.

For **rotated** latitude-longitude coordinates (as for LBCODE=101), the horizontal coordinates differ only slightly –

- The names are 'grid_latitude' and 'grid_longitude'.
- The coord_system is a *iris.coord_systems.RotatedGeogCS*, created with a pole defined by BPLAT, BPLON.

For example:

```
>>> # Load a PP field.
... fname = iris.sample_data_path('air_temp.pp')
>>> fields_iter = iris.fileformats.pp.load(fname)
>>> field = next(fields_iter)
>>>
>>> # Show grid details and first 5 longitude values.
>>> print(' '.join(str(_) for _ in (field.lbcode, field.lbnpt, field.bzx,
...                                 field.bdx)))
1 96 -3.749999 3.749999
>>> print(field.bzx + field.bdx * np.arange(1, 6))
[ 0.    3.75  7.5  11.25 15.  ]
>>>
>>> # Show Iris equivalent information.
... cube = iris.load_cube(fname)
>>> print(cube.coord('longitude').points[:5])
[ 0.    3.75  7.5  11.25 15.  ]
```

Note: Note that in Iris (as in CF) there is no special distinction between "regular" and "irregular" coordinates. Thus on saving, X and Y extra data sections are written only if the actual values are unevenly spaced.

29.1.3 Phenomenon Identification

UM Field elements LBFC, LBUSER4 (aka "stashcode"), LBUSER7 (aka "model code")

Cube components `cube.standard_name`, `cube.units`, `cube.attributes['STASH']`

Details

This information is normally encoded in the cube `standard_name` property. Iris identifies the stash section and item codes from LBUSER4 and the model code in LBUSER7, and compares these against a list of phenomenon types with known CF translations. If the stashcode is recognised, it then defines the appropriate `standard_name` and `units` properties of the cube (i.e. *iris.cube.Cube.standard_name* and *iris.cube.Cube.units*).

Where any parts of the stash information are outside the valid range, Iris will instead attempt to interpret LBFC, for which a set of known translations is also stored. This is often the case for fieldsfiles, where LBUSER4 is frequently left as 0.

In all cases, Iris also constructs a *STASH* item to identify the phenomenon, which is stored as a cube attribute named `STASH`. This preserves the original STASH coding (as standard name translation is not always one-to-one), and can be used when no standard_name translation is identified (for example, to load only certain stashcodes with a constraint – see example at *Load constraint examples*).

For example:

```
>>> # Show PPfield phenomenon details.
>>> print(field.lbuser[3])
16203
>>> print(field.lbuser[6])
1
>>>
>>>
>>> # Show Iris equivalents.
>>> print(cube.standard_name)
```

(continues on next page)

(continued from previous page)

```
air_temperature
>>> print(cube.units)
K
>>> print(cube.attributes['STASH'])
m01s16i203
```

Note: On saving data, no attempt is made to translate a cube standard_name into a STASH code, but any attached 'STASH' attribute will be stored into the LBUSER4 and LBUSER7 elements.

29.1.4 Vertical Coordinates

UM Field elements LBVC, LBLEV, BRSVD1 (aka "bulev"), BRSVD2 (aka "bhulev"), BLEV, BRLEV, BHLEV, BHRLEV

Cube components for height levels : coordinate `height`

for pressure levels : coordinate `pressure`

for hybrid height levels :

- coordinates `model_level_number`, `sigma`, `level_height`, `altitude`
- `cube.aux_factories()[0].orography`

for hybrid pressure levels :

- coordinates `model_level_number`, `sigma`, `level_pressure`, `air_pressure`
- `cube.aux_factories()[0].surface_air_pressure`

Details

Several vertical coordinate forms are supported, according to different values of LBVC. The commonest ones are:

- lbvc=1 : height levels
- lbvc=8 : pressure levels
- lbvc=65 : hybrid height

In all these cases, vertical coordinates are created, with points and bounds values taken from the appropriate header elements. In the raw cubes, each vertical coordinate is just a single value, but multiple values will usually occur. The subsequent merge operation will then convert these into multiple-valued coordinates, and create a new vertical data dimension (i.e. a "Z" axis) which they map onto.

For height levels (LBVC=1): A `height` coordinate is created. This has units 'm', points from BLEV, and no bounds. When there are multiple vertical levels, this will become a dimension coordinate mapping to the vertical dimension.

For pressure levels (LBVC=8): A `pressure` coordinate is created. This has units 'hPa', points from BLEV, and no bounds. When there are multiple vertical levels, this will become a dimension coordinate mapping a vertical dimension.

For hybrid height levels (LBVC=65): Three basic vertical coordinates are created:

- `model_level` is dimensionless, with points from LBLEV and no bounds.
- `sigma` is dimensionless, with points from BHLEV and bounds from BHRLEV and BHULEV.
- `level_height` has units of 'm', points from BLEV and bounds from BRLEV and BULEV.

Also in this case, a *HybridHeightFactory* is created, which references the 'level_height' and 'sigma' coordinates. Following raw cube merging, an extra load stage occurs where the attached *HybridHeightFactory* is called to manufacture a new `altitude` coordinate:

- The altitude coordinate is 3D, mapping to the 2 horizontal dimensions *and* the new vertical dimension.
- Its units are 'm'.
- Its points are calculated from those of the 'level_height' and 'sigma' coordinates, and an orography field. If 'sigma' and 'level_height' possess bounds, then bounds are also created for 'altitude'.

To make the altitude coordinate, there must be an orography field present in the load sources. This is a surface altitude reference field, identified (by stashcode) during the main loading operation, and recorded for later use in the hybrid height calculation. If it is absent, a warning message is printed, and no altitude coordinate is produced.

Note that on merging hybrid height data into a cube, only the 'model_level' coordinate becomes a dimension coordinate: The other vertical coordinates remain as auxiliary coordinates, because they may be (variously) multidimensional or non-monotonic.

See an example printout of a hybrid height cube, *here*:

Notice that this contains all of the above coordinates – 'model_level_number', 'sigma', 'level_height' and the derived 'altitude'.

Note: Hybrid pressure levels can also be handled (for LBVC=9). Without going into details, the mechanism is very similar to that for hybrid height: it produces basic coordinates 'model_level_number', 'sigma' and 'level_pressure', and a manufactured 3D 'air_pressure' coordinate.

29.1.5 Time Information

UM Field elements

- "T1" (i.e. LBYR, LBMON, LBDAT, LBHR, LBMIN, LBDAY/LBSEC),
- "T2" (i.e. LBYRD, LBMOND, LBDATD, LBHRD, LBMIND, LBDAYD/LBSECD),
- LBTIM, LBFT

Cube components coordinates `time`, `forecast_reference_time`, `forecast_period`

Details

In Iris (as in CF) times and time intervals are both expressed as simple numbers, following the approach of the UDUNITS project. These values are stored as cube coordinates, where the scaling and calendar information is contained in the *units* property.

- The units of a time interval (e.g. 'forecast_period'), can be 'seconds' or a simple derived unit such as 'hours' or 'days' – but it does not contain a calendar, so 'months' or 'years' are not valid.
- The units of calendar-based times (including 'time' and 'forecast_reference_time'), are of the general form "<time-unit> since <base-date>", interpreted according to the unit's `calendar` property. The base date for this is always 1st Jan 1970 (times before this are represented as negative values).

The units.calendar property of time coordinates is set from the lowest decimal digit of LBTIM, known as LBTIM.IC. Note that the non-gregorian calendars (e.g. 360-day 'model' calendar) are defined in CF, not udunits.

There are a number of different time encoding methods used in UM data, but the important distinctions are controlled by the next-to-lowest decimal digit of LBTIM, known as "LBTIM.IB". The most common cases are as follows:

Data at a single measurement timepoint (LBTIM.IB=0): A single `time` coordinate is created, with points taken from T1 values. It has no bounds, units of 'hours since 1970-01-01 00:00:00' and a calendar defined according to LBTIM.IC.

Values forecast from T2, valid at T1 (LBTIM.IB=1): Coordinates `time` and ``forecast_reference_time`` are created from the T1 and T2 values, respectively. These have no bounds, and units of 'hours since 1970-01-01 00:00:00', with the appropriate calendar. A `forecast_period` coordinate is also created, with values T1-T2, no bounds and units of 'hours'.

Time mean values between T1 and T2 (LBTIM.IB=2): The time coordinates `time`, `forecast_reference_times` and `forecast_reference_time`, are all present, as in the previous case. In this case, however, the 'time' and 'forecast_period' coordinates also have associated bounds: The 'time' bounds are from T1 to T2, and the 'forecast_period' bounds are from "LBFT - (T2-T1)" to "LBFT".

Note that, in those more complex cases where the input defines all three of the 'time', 'forecast_reference_time' and 'forecast_period' values, any or all of these may become dimensions of the resulting data cube. This will depend on the values actually present in the source fields for each of the elements.

See an example printout of a forecast data cube, *here* :

> Notice that this example contains all of the above coordinates – 'time', 'forecast_period' and 'forecast_reference_time'. In this case the data are forecasts, so 'time' is a dimension, 'forecast_period' varies with time and 'forecast_reference_time' is a constant.

29.1.6 Statistical Measures

UM Field elements LBPROC, LBTIM

Cube components `cube.cell_methods`

Details

Where a field contains statistically processed data, Iris will add an appropriate *`iris.coords.CellMethod`* to the cube, representing the aggregation operation which was performed.

This is implemented for certain binary flag bits within the LBPROC element value. For example:

- **time mean, when (LBPROC & 128):** Cube has a cell_method of the form "CellMethod('mean', 'time').
- **time period minimum value, when (LBPROC & 4096):** Cube has a cell_method of the form "CellMethod('minimum', 'time').
- **time period maximum value, when (LBPROC & 8192):** Cube has a cell_method of the form "CellMethod('maximum', 'time').

In all these cases, if the field LBTIM is also set to denote a time aggregate field (i.e. "LBTIM.IB=2", see above *Time Information*), then the second-to-last digit of LBTIM, aka "LBTIM.IA" may also be non-zero, in which case this indicates the aggregation time-interval. In that case, the cell-method *`intervals`* attribute is also set to this many hours.

For example:

```
>>> # Show stats metadata in a test PP field.
... fname = iris.sample_data_path('pre-industrial.pp')
>>> eg_field = next(iris.fileformats.pp.load(fname))
>>> print(eg_field.lbtim)
622
>>> print(eg_field.lbproc)
128
>>>
```

(continues on next page)

(continued from previous page)

```
>>> # Print out the Iris equivalent information.
>>> print(iris.load_cube(fname).cell_methods)
(CellMethod(method='mean', coord_names=('time',), intervals=('6 hour',),␣
↪comments=()),)
```

29.1.7 Other Metadata

LBRSVD4

If non-zero, this is interpreted as an ensemble number. This produces a cube scalar coordinate named 'realization' (as defined in the CF conventions).

LBUSER5

If non-zero, this is interpreted as a 'pseudo_level' number. This produces a cube scalar coordinate named 'pseudo_level'. In the UM documentation LBUSER5 is also sometimes referred to as LBPLEV.

29.2 Missing Data Handling in Iris

This document provides a brief overview of how Iris handles missing data values when datasets are loaded as cubes, and when cubes are saved or modified.

A missing data value, or fill-value, defines the value used within a dataset to indicate that data point is missing or not set. This value is included as part of a dataset's metadata.

For example, in a gridded global ocean dataset, no data values will be recorded over land, so land points will be missing data. In such a case, land points could be indicated by being set to the dataset's missing data value.

29.2.1 Loading

On load, any fill-value or missing data value defined in the loaded dataset should be used as the `fill_value` of the NumPy masked array data attribute of the *Cube*. This will only appear when the cube's data is realised.

29.2.2 Saving

On save, the fill-value of a cube's masked data array is **not** used in saving data. Instead, Iris always uses the default fill-value for the fileformat, *except* when a fill-value is specified by the user via a fileformat-specific saver.

For example:

```
>>> iris.save(my_cube, 'my_file.nc', fill_value=-99999)
```

Note: Not all savers accept the `fill_value` keyword argument.

Iris will check for and issue warnings of fill-value 'collisions'. This basically means that whenever there are unmasked values that would read back as masked, we issue a warning and suggest a workaround.

This will occur in the following cases:

- where masked data contains *unmasked* points matching the fill-value, or
- where unmasked data contains the fill-value (either the format-specific default fill-value, or a fill-value specified by the user in the save call).

NetCDF

NetCDF is a special case, because all ordinary variable data is "potentially masked", owing to the use of default fill values. The default fill-value used depends on the type of the variable data.

The exceptions to this are:

- One-byte values are not masked unless the variable has an explicit `_FillValue` attribute. That is, there is no default fill-value for `byte` types in NetCDF.
- Data may be tagged with a `_NoFill` attribute. This is not currently officially documented or widely implemented.
- Small integers create problems by *not* having the exemption applied to byte data. Thus, in principle, `int32` data cannot use the full range of 2**16 valid values.

29.2.3 Merging

Merged data should have a fill-value equal to that of the components, if they all have the same fill-value. If the components have differing fill-values, a default fill-value will be used instead.

29.2.4 Other Operations

Other operations, such as *Cube* arithmetic operations, generally produce output with a default (NumPy) fill-value. That is, these operations ignore the fill-values of the input(s) to the operation.

CHAPTER

THIRTY

IRIS COPYRIGHT, LICENSING AND CONTRIBUTORS

30.1 Iris Code

All Iris source code, unless explicitly stated, is `Copyright Iris contributors` and is licensed under the **GNU Lesser General Public License** as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. You should find all source files with the following header:

Code License

Copyright Iris contributors

This file is part of Iris and is released under the LGPL license. See COPYING and COPYING.LESSER in the root of the repository for full licensing details.

30.2 Iris Documentation and Examples

All documentation, examples and sample data found on this website and in source repository are licensed under the UK's Open Government Licence:

Documentation, example and data license

(C) British Crown Copyright 2010 - 2021

You may use and re-use the information featured on this website (not including logos) free of charge in any format or medium, under the terms of the Open Government Licence. We encourage users to establish hypertext links to this website.

Any email enquiries regarding the use and re-use of this information resource should be sent to: psi@nationalarchives.gsi.gov.uk.

BIBLIOGRAPHY

[Jackson] Jackson, M. 2012. How to cite and describe software. Accessed 06-03-2013.

PYTHON MODULE INDEX

.

iris.analysis, 256
iris.analysis.calculus, 239
iris.analysis.cartography, 241
iris.analysis.geometry, 248
iris.analysis.maths, 248
iris.analysis.stats, 253
iris.analysis.trajectory, 254
iris.aux_factory, 272
iris.common, 309
iris.common.lenient, 287
iris.common.metadata, 288
iris.common.mixin, 301
iris.common.resolve, 302
iris.config, 309
iris.coord_categorisation, 311
iris.coord_systems, 314
iris.coords, 325
iris.cube, 351
iris.exceptions, 373
iris.experimental, 384
iris.experimental.animate, 377
iris.experimental.equalise_cubes, 378
iris.experimental.regrid, 378
iris.experimental.regrid_conservative, 382
iris.experimental.representation, 382
iris.experimental.stratify, 383
iris.experimental.ugrid, 384
iris.fileformats, 428
iris.fileformats.abf, 384
iris.fileformats.cf, 385
iris.fileformats.dot, 402
iris.fileformats.name, 402
iris.fileformats.name_loaders, 403
iris.fileformats.netcdf, 405
iris.fileformats.nimrod, 412
iris.fileformats.nimrod_load_rules, 413
iris.fileformats.pp, 413
iris.fileformats.pp_load_rules, 419
iris.fileformats.pp_save_rules, 419
iris.fileformats.rules, 419
iris.fileformats.um, 423
iris.fileformats.um_cf_map, 428
iris.io, 431
iris.io.format_picker, 428
iris.iterate, 435
iris.palette, 435
iris.pandas, 437
iris.plot, 438
iris.quickplot, 444
iris.std_names, 446
iris.symbols, 447
iris.time, 447
iris.util, 448

d

documenting.docstrings_attribute, 217
documenting.docstrings_sample_routine, 215

i

iris, 459

INDEX

Symbols

__binary_operator__() (*iris.coords.AncillaryVariable method*), 326
__binary_operator__() (*iris.coords.AuxCoord method*), 328
__binary_operator__() (*iris.coords.CellMeasure method*), 335
__binary_operator__() (*iris.coords.Coord method*), 338
__binary_operator__() (*iris.coords.DimCoord method*), 345
__call__() (*iris.analysis.maths.IFunc method*), 253
__call__() (*iris.common.resolve.Resolve method*), 304
__call__() (*iris.palette.SymmetricNormalize method*), 436
__common_cmp__() (*iris.coords.Cell method*), 334
__copy__() (*iris.cube.Cube method*), 353
__deepcopy__() (*iris.coords.DimCoord method*), 346
__eq__() (*iris.common.metadata.AncillaryVariableMetadata method*), 288
__eq__() (*iris.common.metadata.BaseMetadata method*), 290
__eq__() (*iris.common.metadata.CellMeasureMetadata method*), 292
__eq__() (*iris.common.metadata.CoordMetadata method*), 294
__eq__() (*iris.common.metadata.CubeMetadata method*), 296
__eq__() (*iris.common.metadata.DimCoordMetadata method*), 298
__eq__() (*iris.coords.Cell method*), 334
__exit__() (*iris.fileformats.netcdf.Saver method*), 410
__getattr__() (*iris.fileformats.pp.PPField method*), 415
__getitem__() (*iris.coords.AncillaryVariable method*), 326
__getitem__() (*iris.coords.AuxCoord method*), 329
__getitem__() (*iris.coords.CellMeasure method*), 335
__getitem__() (*iris.coords.Coord method*), 339
__getitem__() (*iris.cube.Cube method*), 353
__getitem__() (*iris.cube.CubeList method*), 368
__getslice__() (*iris.cube.CubeList method*), 368
__new__() (*iris.coords.Cell static method*), 334
__new__() (*iris.coords.CoordExtent static method*), 344
__new__() (*iris.cube.CubeList static method*), 368
__new__() (*iris.fileformats.pp.STASH static method*), 418
__new__() (*iris.fileformats.rules.Loader static method*), 422
__repr__() (*iris.cube.CubeList method*), 368
__repr__() (*iris.fileformats.pp.PPField method*), 415
__str__() (*iris.coords.CellMethod method*), 337
__str__() (*iris.cube.CubeList method*), 368

A

a (*documenting.docstrings_attribute.ExampleClass attribute*), 217
ABFField (*class in iris.fileformats.abf*), 385
abs() (*in module iris.analysis.maths*), 249
add() (*in module iris.analysis.maths*), 249
add() (*iris.fileformats.netcdf.OrderedAddableList method*), 408
add_ancillary_variable() (*iris.cube.Cube method*), 353
add_aux_coord() (*iris.cube.Cube method*), 353
add_aux_factory() (*iris.cube.Cube method*), 353
add_categorised_coord() (*in module iris.coord_categorisation*), 311
add_cell_measure() (*iris.cube.Cube method*), 353
add_cell_method() (*iris.cube.Cube method*), 353
add_cube() (*iris.fileformats.rules.ConcreteReferenceTarget method*), 420
add_day_of_month() (*in module iris.coord_categorisation*), 312
add_day_of_year() (*in module iris.coord_categorisation*), 312
add_dim_coord() (*iris.cube.Cube method*), 354
add_formula_term() (*iris.fileformats.cf.CFAncillaryDataVariable method*), 386
add_formula_term() (*iris.fileformats.cf.CFAuxiliaryCoordinateVariable method*), 387
add_formula_term() (*iris.fileformats.cf.CFBoundaryVariable method*), 389

add_formula_term() (*iris.fileformats.cf.CFClimatologyVariable method*), 390
add_formula_term() (*iris.fileformats.cf.CFCoordinateVariable method*), 392
add_formula_term() (*iris.fileformats.cf.CFDataVariable method*), 393
add_formula_term() (*iris.fileformats.cf.CFGridMappingVariable method*), 395
add_formula_term() (*iris.fileformats.cf.CFLabelVariable method*), 397
add_formula_term() (*iris.fileformats.cf.CFMeasureVariable method*), 399
add_formula_term() (*iris.fileformats.cf.CFVariable method*), 400
add_hour() (*in module iris.coord_categorisation*), 312
add_month() (*in module iris.coord_categorisation*), 312
add_month_fullname() (*in module iris.coord_categorisation*), 312
add_month_number() (*in module iris.coord_categorisation*), 312
add_saver() (*in module iris.io*), 432
add_season() (*in module iris.coord_categorisation*), 312
add_season_membership() (*in module iris.coord_categorisation*), 312
add_season_number() (*in module iris.coord_categorisation*), 313
add_season_year() (*in module iris.coord_categorisation*), 313
add_spec() (*iris.io.format_picker.FormatAgent method*), 430
add_weekday() (*in module iris.coord_categorisation*), 313
add_weekday_fullname() (*in module iris.coord_categorisation*), 313
add_weekday_number() (*in module iris.coord_categorisation*), 313
add_year() (*in module iris.coord_categorisation*), 314
aggregate() (*iris.analysis.Aggregator method*), 263
aggregate() (*iris.analysis.WeightedAggregator method*), 265
aggregate_shape() (*iris.analysis.Aggregator method*), 264
aggregate_shape() (*iris.analysis.WeightedAggregator method*), 266
aggregated_by() (*iris.cube.Cube method*), 354
Aggregator (*class in iris.analysis*), 263
AlbersEqualArea (*class in iris.coord_systems*), 314
ancillary_variable() (*iris.cube.Cube method*), 355
ancillary_variable_dims() (*iris.cube.Cube method*), 355
ancillary_variables (*iris.fileformats.cf.CFGroup property*), 396
ancillary_variables() (*iris.cube.Cube method*), 355
AncillaryVariable (*class in iris.coords*), 326
AncillaryVariableMetadata (*class in iris.common.metadata*), 288
AncillaryVariableNotFoundError (*class in iris.exceptions*), 374
animate() (*in module iris.experimental.animate*), 377
append() (*iris.cube.CubeList method*), 368
append() (*iris.fileformats.netcdf.CFNameCoordMap method*), 408
append() (*iris.fileformats.netcdf.OrderedAddableList method*), 409
apply_ufunc() (*in module iris.analysis.maths*), 249
approx_equal() (*in module iris.util*), 449
area_weights() (*in module iris.analysis.cartography*), 241
AreaWeighted (*class in iris.analysis*), 269
args (*iris.exceptions.AncillaryVariableNotFoundError attribute*), 374
args (*iris.exceptions.CellMeasureNotFoundError attribute*), 374
args (*iris.exceptions.ConcatenateError attribute*), 374
args (*iris.exceptions.ConstraintMismatchError attribute*), 374
args (*iris.exceptions.CoordinateCollapseError attribute*), 375
args (*iris.exceptions.CoordinateMultiDimError attribute*), 375
args (*iris.exceptions.CoordinateNotFoundError attribute*), 375
args (*iris.exceptions.CoordinateNotRegularError attribute*), 375
args (*iris.exceptions.DuplicateDataError attribute*), 375
args (*iris.exceptions.IgnoreCubeException attribute*), 375
args (*iris.exceptions.InvalidCubeError attribute*), 376
args (*iris.exceptions.IrisError attribute*), 376
args (*iris.exceptions.LazyAggregatorError attribute*), 376
args (*iris.exceptions.MergeError attribute*), 376
args (*iris.exceptions.NotYetImplementedError attribute*), 376
args (*iris.exceptions.TranslationError attribute*), 377
args (*iris.exceptions.UnitConversionError attribute*), 377
args (*iris.fileformats.netcdf.UnknownCellMethodWarning attribute*), 412
args (*iris.fileformats.rules.Factory attribute*), 422
args (*iris.IrisDeprecation attribute*), 465
array_equal() (*in module iris.util*), 449
as_cartopy_crs() (*iris.coord_systems.AlbersEqualArea method*), 315
as_cartopy_crs() (*iris.coord_systems.CoordSystem method*), 315
as_cartopy_crs() (*iris.coord_systems.GeogCS*

method), 316
as_cartopy_crs() (*iris.coord_systems.Geostationary method*), 317
as_cartopy_crs() (*iris.coord_systems.LambertAzimuthalEqualArea method*), 318
as_cartopy_crs() (*iris.coord_systems.LambertConformal method*), 319
as_cartopy_crs() (*iris.coord_systems.Mercator method*), 320
as_cartopy_crs() (*iris.coord_systems.Orthographic method*), 321
as_cartopy_crs() (*iris.coord_systems.OSGB method*), 320
as_cartopy_crs() (*iris.coord_systems.RotatedGeogCS method*), 322
as_cartopy_crs() (*iris.coord_systems.Stereographic method*), 323
as_cartopy_crs() (*iris.coord_systems.TransverseMercator method*), 324
as_cartopy_crs() (*iris.coord_systems.VerticalPerspective method*), 325
as_cartopy_globe() (*iris.coord_systems.GeogCS method*), 316
as_cartopy_projection() (*iris.coord_systems.AlbersEqualArea method*), 315
as_cartopy_projection() (*iris.coord_systems.CoordSystem method*), 315
as_cartopy_projection() (*iris.coord_systems.GeogCS method*), 316
as_cartopy_projection() (*iris.coord_systems.Geostationary method*), 317
as_cartopy_projection() (*iris.coord_systems.LambertAzimuthalEqualArea method*), 318
as_cartopy_projection() (*iris.coord_systems.LambertConformal method*), 319
as_cartopy_projection() (*iris.coord_systems.Mercator method*), 320
as_cartopy_projection() (*iris.coord_systems.Orthographic method*), 321
as_cartopy_projection() (*iris.coord_systems.OSGB method*), 320
as_cartopy_projection() (*iris.coord_systems.RotatedGeogCS method*), 322
as_cartopy_projection() (*iris.coord_systems.Stereographic method*), 323
as_cartopy_projection() (*iris.coord_systems.TransverseMercator method*), 324
as_cartopy_projection() (*iris.coord_systems.VerticalPerspective method*), 325
as_compatible_shape() (*in module iris.util*), 449
as_cube() (*in module iris.pandas*), 437
as_cube() (*iris.fileformats.rules.ConcreteReferenceTarget method*), 421
as_data_frame() (*in module iris.pandas*), 438
as_fields() (*in module iris.fileformats.pp*), 416
as_series() (*in module iris.pandas*), 438
AtmosphereSigmaFactory (*class in iris.aux_factory*), 273
AttributeConstraint (*class in iris*), 463
attributes (*iris.aux_factory.AtmosphereSigmaFactory property*), 274
attributes (*iris.aux_factory.AuxCoordFactory property*), 275
attributes (*iris.aux_factory.HybridHeightFactory property*), 277
attributes (*iris.aux_factory.HybridPressureFactory property*), 278
attributes (*iris.aux_factory.OceanSFactory property*), 280
attributes (*iris.aux_factory.OceanSg1Factory property*), 282
attributes (*iris.aux_factory.OceanSg2Factory property*), 283
attributes (*iris.aux_factory.OceanSigmaFactory property*), 285
attributes (*iris.aux_factory.OceanSigmaZFactory property*), 286
attributes (*iris.common.metadata.AncillaryVariableMetadata attribute*), 290
attributes (*iris.common.metadata.BaseMetadata attribute*), 292
attributes (*iris.common.metadata.CellMeasureMetadata attribute*), 294
attributes (*iris.common.metadata.CoordMetadata attribute*), 296
attributes (*iris.common.metadata.CubeMetadata attribute*), 298
attributes (*iris.common.metadata.DimCoordMetadata attribute*), 300
attributes (*iris.common.mixin.CFVariableMixin property*), 301
attributes (*iris.coords.AncillaryVariable property*), 327
attributes (*iris.coords.AuxCoord property*), 333
attributes (*iris.coords.CellMeasure property*), 337
attributes (*iris.coords.Coord property*), 343
attributes (*iris.coords.DimCoord property*), 350
attributes (*iris.cube.Cube property*), 366

attributes (*iris.fileformats.rules.ConversionMetadata attribute*), 421
auto_palette() (*in module iris.palette*), 435
autoscale() (*iris.palette.SymmetricNormalize method*), 437
autoscale_None() (*iris.palette.SymmetricNormalize method*), 437
aux_coords (*iris.cube.Cube property*), 367
aux_coords_and_dims (*iris.fileformats.rules.ConversionMetadata attribute*), 421
aux_factories (*iris.cube.Cube property*), 367
aux_factory() (*in module iris.fileformats.rules*), 420
aux_factory() (*iris.cube.Cube method*), 355
AuxCoord (*class in iris.coords*), 328
AuxCoordFactory (*class in iris.aux_factory*), 274
auxiliary_coordinates (*iris.fileformats.cf.CFGroup property*), 396

B

b (*documenting.docstrings_attribute.ExampleClass attribute*), 217
barbs() (*in module iris.plot*), 439
BaseMetadata (*class in iris.common.metadata*), 290
between() (*in module iris.util*), 450
bmdi (*iris.fileformats.um.FieldCollation property*), 427
bound (*iris.coords.Cell attribute*), 335
bounds (*iris.coords.AuxCoord property*), 333
bounds (*iris.coords.Coord property*), 343
bounds (*iris.coords.DimCoord property*), 350
bounds (*iris.fileformats.cf.CFGroup property*), 396
bounds_dtype (*iris.coords.AuxCoord property*), 333
bounds_dtype (*iris.coords.Coord property*), 343
bounds_dtype (*iris.coords.DimCoord property*), 350
broadcast_to_shape() (*in module iris.util*), 450

C

calendar (*iris.fileformats.pp.PPField property*), 416
call_func (*iris.analysis.Aggregator attribute*), 265
call_func (*iris.analysis.WeightedAggregator attribute*), 267
category_common (*iris.common.resolve.Resolve attribute*), 305
Cell (*class in iris.coords*), 334
cell() (*iris.coords.AuxCoord method*), 329
cell() (*iris.coords.Coord method*), 339
cell() (*iris.coords.DimCoord method*), 346
cell_measure() (*iris.cube.Cube method*), 355
cell_measure_dims() (*iris.cube.Cube method*), 356
cell_measures (*iris.fileformats.cf.CFGroup property*), 396
cell_measures() (*iris.cube.Cube method*), 356
cell_method (*iris.analysis.Aggregator attribute*), 265
cell_method (*iris.analysis.WeightedAggregator attribute*), 267
cell_methods (*iris.common.metadata.CubeMetadata attribute*), 298
cell_methods (*iris.cube.Cube property*), 367
cell_methods (*iris.fileformats.rules.ConversionMetadata attribute*), 421
CellMeasure (*class in iris.coords*), 335
CellMeasureMetadata (*class in iris.common.metadata*), 292
CellMeasureNotFoundError (*class in iris.exceptions*), 374
CellMethod (*class in iris.coords*), 337
cells() (*iris.coords.AuxCoord method*), 329
cells() (*iris.coords.Coord method*), 339
cells() (*iris.coords.DimCoord method*), 346
central_lat (*iris.coord_systems.LambertConformal attribute*), 319
central_lat (*iris.coord_systems.Stereographic attribute*), 323
central_lon (*iris.coord_systems.LambertConformal attribute*), 319
central_lon (*iris.coord_systems.Stereographic attribute*), 323
cf_attrs() (*iris.fileformats.cf.CFAncillaryDataVariable method*), 386
cf_attrs() (*iris.fileformats.cf.CFAuxiliaryCoordinateVariable method*), 387
cf_attrs() (*iris.fileformats.cf.CFBoundaryVariable method*), 389
cf_attrs() (*iris.fileformats.cf.CFClimatologyVariable method*), 391
cf_attrs() (*iris.fileformats.cf.CFCoordinateVariable method*), 392
cf_attrs() (*iris.fileformats.cf.CFDataVariable method*), 393
cf_attrs() (*iris.fileformats.cf.CFGridMappingVariable method*), 395
cf_attrs() (*iris.fileformats.cf.CFLabelVariable method*), 397
cf_attrs() (*iris.fileformats.cf.CFMeasureVariable method*), 399
cf_attrs() (*iris.fileformats.cf.CFVariable method*), 400
cf_attrs_ignored() (*iris.fileformats.cf.CFAncillaryDataVariable method*), 386
cf_attrs_ignored() (*iris.fileformats.cf.CFAuxiliaryCoordinateVariable method*), 388
cf_attrs_ignored() (*iris.fileformats.cf.CFBoundaryVariable method*), 389
cf_attrs_ignored() (*iris.fileformats.cf.CFClimatologyVariable method*), 391
cf_attrs_ignored() (*iris.fileformats.cf.CFCoordinateVariable method*), 392
cf_attrs_ignored() (*iris.fileformats.cf.CFDataVariable*

method), 393
cf_attrs_ignored() (iris.fileformats.cf.CFGridMappingVariable method), 395
cf_attrs_ignored() (iris.fileformats.cf.CFLabelVariable method), 397
cf_attrs_ignored() (iris.fileformats.cf.CFMeasureVariable method), 399
cf_attrs_ignored() (iris.fileformats.cf.CFVariable method), 400
cf_attrs_reset() (iris.fileformats.cf.CFAncillaryDataVariable method), 386
cf_attrs_reset() (iris.fileformats.cf.CFAuxiliaryCoordinateVariable method), 388
cf_attrs_reset() (iris.fileformats.cf.CFBoundaryVariable method), 389
cf_attrs_reset() (iris.fileformats.cf.CFClimatologyVariable method), 391
cf_attrs_reset() (iris.fileformats.cf.CFCoordinateVariable method), 392
cf_attrs_reset() (iris.fileformats.cf.CFDataVariable method), 393
cf_attrs_reset() (iris.fileformats.cf.CFGridMappingVariable method), 395
cf_attrs_reset() (iris.fileformats.cf.CFLabelVariable method), 397
cf_attrs_reset() (iris.fileformats.cf.CFMeasureVariable method), 399
cf_attrs_reset() (iris.fileformats.cf.CFVariable method), 400
cf_attrs_unused() (iris.fileformats.cf.CFAncillaryDataVariable method), 386
cf_attrs_unused() (iris.fileformats.cf.CFAuxiliaryCoordinateVariable method), 388
cf_attrs_unused() (iris.fileformats.cf.CFBoundaryVariable method), 389
cf_attrs_unused() (iris.fileformats.cf.CFClimatologyVariable method), 391
cf_attrs_unused() (iris.fileformats.cf.CFCoordinateVariable method), 392
cf_attrs_unused() (iris.fileformats.cf.CFDataVariable method), 393
cf_attrs_unused() (iris.fileformats.cf.CFGridMappingVariable method), 395
cf_attrs_unused() (iris.fileformats.cf.CFLabelVariable method), 397
cf_attrs_unused() (iris.fileformats.cf.CFMeasureVariable method), 399
cf_attrs_unused() (iris.fileformats.cf.CFVariable method), 400
cf_attrs_used() (iris.fileformats.cf.CFAncillaryDataVariable method), 386
cf_attrs_used() (iris.fileformats.cf.CFAuxiliaryCoordinateVariable method), 388
cf_attrs_used() (iris.fileformats.cf.CFBoundaryVariable method), 389
cf_attrs_used() (iris.fileformats.cf.CFClimatologyVariable method), 391
cf_attrs_used() (iris.fileformats.cf.CFCoordinateVariable method), 392
cf_attrs_used() (iris.fileformats.cf.CFDataVariable method), 393
cf_attrs_used() (iris.fileformats.cf.CFGridMappingVariable method), 395
cf_attrs_used() (iris.fileformats.cf.CFLabelVariable method), 397
cf_attrs_used() (iris.fileformats.cf.CFMeasureVariable method), 399
cf_attrs_used() (iris.fileformats.cf.CFVariable method), 400
cf_data (iris.fileformats.cf.CFAncillaryDataVariable attribute), 387
cf_data (iris.fileformats.cf.CFAuxiliaryCoordinateVariable attribute), 388
cf_data (iris.fileformats.cf.CFBoundaryVariable attribute), 390
cf_data (iris.fileformats.cf.CFClimatologyVariable attribute), 391
cf_data (iris.fileformats.cf.CFCoordinateVariable attribute), 393
cf_data (iris.fileformats.cf.CFDataVariable attribute), 394
cf_data (iris.fileformats.cf.CFGridMappingVariable attribute), 395
cf_data (iris.fileformats.cf.CFLabelVariable attribute), 398
cf_data (iris.fileformats.cf.CFMeasureVariable attribute), 399
cf_data (iris.fileformats.cf.CFVariable attribute), 401
cf_group (iris.fileformats.cf.CFAncillaryDataVariable attribute), 387
cf_group (iris.fileformats.cf.CFAuxiliaryCoordinateVariable attribute), 388
cf_group (iris.fileformats.cf.CFBoundaryVariable attribute), 390
cf_group (iris.fileformats.cf.CFClimatologyVariable attribute), 391
cf_group (iris.fileformats.cf.CFCoordinateVariable attribute), 393
cf_group (iris.fileformats.cf.CFDataVariable attribute), 394
cf_group (iris.fileformats.cf.CFGridMappingVariable attribute), 395
cf_group (iris.fileformats.cf.CFLabelVariable attribute), 398
cf_group (iris.fileformats.cf.CFMeasureVariable attribute), 399
cf_group (iris.fileformats.cf.CFReader attribute), 400
cf_group (iris.fileformats.cf.CFVariable attribute), 401

cf_identity (*iris.fileformats.cf.CFAncillaryDataVariable attribute*), 387
cf_identity (*iris.fileformats.cf.CFAuxiliaryCoordinateVariable attribute*), 388
cf_identity (*iris.fileformats.cf.CFBoundaryVariable attribute*), 390
cf_identity (*iris.fileformats.cf.CFClimatologyVariable attribute*), 391
cf_identity (*iris.fileformats.cf.CFCoordinateVariable attribute*), 393
cf_identity (*iris.fileformats.cf.CFDataVariable attribute*), 394
cf_identity (*iris.fileformats.cf.CFGridMappingVariable attribute*), 395
cf_identity (*iris.fileformats.cf.CFLabelVariable attribute*), 398
cf_identity (*iris.fileformats.cf.CFMeasureVariable attribute*), 400
cf_identity (*iris.fileformats.cf.CFVariable attribute*), 401
cf_label_data() (*iris.fileformats.cf.CFLabelVariable method*), 397
cf_label_dimensions() (*iris.fileformats.cf.CFLabelVariable method*), 398
cf_measure (*iris.fileformats.cf.CFMeasureVariable attribute*), 400
cf_name (*iris.fileformats.cf.CFAncillaryDataVariable attribute*), 387
cf_name (*iris.fileformats.cf.CFAuxiliaryCoordinateVariable attribute*), 388
cf_name (*iris.fileformats.cf.CFBoundaryVariable attribute*), 390
cf_name (*iris.fileformats.cf.CFClimatologyVariable attribute*), 391
cf_name (*iris.fileformats.cf.CFCoordinateVariable attribute*), 393
cf_name (*iris.fileformats.cf.CFDataVariable attribute*), 394
cf_name (*iris.fileformats.cf.CFGridMappingVariable attribute*), 395
cf_name (*iris.fileformats.cf.CFLabelVariable attribute*), 398
cf_name (*iris.fileformats.cf.CFMeasureVariable attribute*), 400
cf_name (*iris.fileformats.cf.CFVariable attribute*), 401
cf_terms_by_root (*iris.fileformats.cf.CFAncillaryDataVariable attribute*), 387
cf_terms_by_root (*iris.fileformats.cf.CFAuxiliaryCoordinateVariable attribute*), 388
cf_terms_by_root (*iris.fileformats.cf.CFBoundaryVariable attribute*), 390
cf_terms_by_root (*iris.fileformats.cf.CFClimatologyVariable attribute*), 391
cf_terms_by_root (*iris.fileformats.cf.CFCoordinateVariable attribute*), 393
cf_terms_by_root (*iris.fileformats.cf.CFDataVariable attribute*), 394
cf_terms_by_root (*iris.fileformats.cf.CFGridMappingVariable attribute*), 396
cf_terms_by_root (*iris.fileformats.cf.CFLabelVariable attribute*), 398
cf_terms_by_root (*iris.fileformats.cf.CFMeasureVariable attribute*), 400
cf_terms_by_root (*iris.fileformats.cf.CFVariable attribute*), 401
cf_valid_var_name() (*iris.fileformats.netcdf.Saver static method*), 410
CFAncillaryDataVariable (*class in iris.fileformats.cf*), 386
CFAuxiliaryCoordinateVariable (*class in iris.fileformats.cf*), 387
CFBoundaryVariable (*class in iris.fileformats.cf*), 389
CFClimatologyVariable (*class in iris.fileformats.cf*), 390
CFCoordinateVariable (*class in iris.fileformats.cf*), 392
CFDataVariable (*class in iris.fileformats.cf*), 393
CFGridMappingVariable (*class in iris.fileformats.cf*), 394
CFGroup (*class in iris.fileformats.cf*), 396
CFLabelVariable (*class in iris.fileformats.cf*), 397
CFMeasureVariable (*class in iris.fileformats.cf*), 399
CFName (*class in iris.fileformats.um_cf_map*), 428
CFNameCoordMap (*class in iris.fileformats.netcdf*), 408
CFReader (*class in iris.fileformats.cf*), 400
CFVariable (*class in iris.fileformats.cf*), 400
CFVariableMixin (*class in iris.common.mixin*), 301
check_attribute_compliance() (*iris.fileformats.netcdf.Saver static method*), 410
circular (*iris.common.metadata.DimCoordMetadata attribute*), 300
circular (*iris.coords.DimCoord property*), 350
citation() (*in module iris.plot*), 439
clear() (*iris.cube.CubeList method*), 368
clear() (*iris.fileformats.cf.CFGroup method*), 396
clear() (*iris.fileformats.netcdf.OrderedAddableList method*), 409
clear_phenomenon_identity() (*in module iris.analysis*), 267
climatological (*iris.aux_factory.AtmosphereSigmaFactory property*), 274
climatological (*iris.aux_factory.AuxCoordFactory property*), 275
climatological (*iris.aux_factory.HybridHeightFactory property*), 277
climatological (*iris.aux_factory.HybridPressureFactory*

property), 279
climatological (*iris.aux_factory.OceanSFactory property*), 280
climatological (*iris.aux_factory.OceanSg1Factory property*), 282
climatological (*iris.aux_factory.OceanSg2Factory property*), 283
climatological (*iris.aux_factory.OceanSigmaFactory property*), 285
climatological (*iris.aux_factory.OceanSigmaZFactory property*), 286
climatological (*iris.common.metadata.CoordMetadata attribute*), 296
climatological (*iris.common.metadata.DimCoordMetadata attribute*), 300
climatological (*iris.coords.AuxCoord property*), 333
climatological (*iris.coords.Coord property*), 343
climatological (*iris.coords.DimCoord property*), 350
climatology (*iris.fileformats.cf.CFGroup property*), 396
clip_string() (*in module iris.util*), 451
CLOUD_COVER (*in module iris.symbols*), 447
cmap_norm() (*in module iris.palette*), 436
collapsed() (*iris.coords.AuxCoord method*), 329
collapsed() (*iris.coords.Coord method*), 339
collapsed() (*iris.coords.DimCoord method*), 346
collapsed() (*iris.cube.Cube method*), 356
column_slices_generator() (*in module iris.util*), 451
combine() (*iris.common.metadata.AncillaryVariableMetadata method*), 289
combine() (*iris.common.metadata.BaseMetadata method*), 290
combine() (*iris.common.metadata.CellMeasureMetadata method*), 292
combine() (*iris.common.metadata.CoordMetadata method*), 294
combine() (*iris.common.metadata.CubeMetadata method*), 296
combine() (*iris.common.metadata.DimCoordMetadata method*), 298
comments (*iris.coords.CellMethod attribute*), 338
concatenate() (*iris.cube.CubeList method*), 368
concatenate_cube() (*iris.cube.CubeList method*), 370
ConcatenateError (*class in iris.exceptions*), 374
ConcreteReferenceTarget (*class in iris.fileformats.rules*), 420
Constraint (*class in iris*), 462
ConstraintMismatchError (*class in iris.exceptions*), 374
contains_point() (*iris.coords.Cell method*), 334
context() (*iris.common.lenient.Lenient method*), 287
context() (*iris.config.NetCDF method*), 310
context() (*iris.Future method*), 465
contiguous_bounds() (*iris.coords.AuxCoord method*), 329
contiguous_bounds() (*iris.coords.Coord method*), 339
contiguous_bounds() (*iris.coords.DimCoord method*), 346
contour() (*in module iris.plot*), 440
contour() (*in module iris.quickplot*), 445
contourf() (*in module iris.plot*), 440
contourf() (*in module iris.quickplot*), 445
ConversionMetadata (*class in iris.fileformats.rules*), 421
convert() (*in module iris.fileformats.pp_load_rules*), 419
convert_units() (*iris.coords.AncillaryVariable method*), 326
convert_units() (*iris.coords.AuxCoord method*), 329
convert_units() (*iris.coords.CellMeasure method*), 335
convert_units() (*iris.coords.Coord method*), 339
convert_units() (*iris.coords.DimCoord method*), 346
convert_units() (*iris.cube.Cube method*), 357
converter (*iris.fileformats.rules.Loader attribute*), 422
Coord (*class in iris.coords*), 338
coord() (*iris.cube.Cube method*), 358
coord() (*iris.fileformats.netcdf.CFNameCoordMap method*), 408
coord_dims() (*iris.cube.Cube method*), 358
coord_names (*iris.coords.CellMethod attribute*), 338
coord_system (*iris.aux_factory.AtmosphereSigmaFactory property*), 274
coord_system (*iris.aux_factory.AuxCoordFactory property*), 275
coord_system (*iris.aux_factory.HybridHeightFactory property*), 277
coord_system (*iris.aux_factory.HybridPressureFactory property*), 279
coord_system (*iris.aux_factory.OceanSFactory property*), 280
coord_system (*iris.aux_factory.OceanSg1Factory property*), 282
coord_system (*iris.aux_factory.OceanSg2Factory property*), 283
coord_system (*iris.aux_factory.OceanSigmaFactory property*), 285
coord_system (*iris.aux_factory.OceanSigmaZFactory property*), 286
coord_system (*iris.common.metadata.CoordMetadata attribute*), 296
coord_system (*iris.common.metadata.DimCoordMetadata attribute*), 300
coord_system (*iris.coords.AuxCoord property*), 333
coord_system (*iris.coords.Coord property*), 343
coord_system (*iris.coords.DimCoord property*), 350
coord_system() (*iris.cube.Cube method*), 358
coord_system() (*iris.fileformats.pp.PPField method*),

415
CoordExtent (*class in iris.coords*), 344
CoordinateCollapseError (*class in iris.exceptions*), 374
CoordinateMultiDimError (*class in iris.exceptions*), 375
CoordinateNotFoundError (*class in iris.exceptions*), 375
CoordinateNotRegularError (*class in iris.exceptions*), 375
coordinates (*iris.fileformats.cf.CFGroup property*), 396
CoordMetadata (*class in iris.common.metadata*), 294
coords (*iris.fileformats.netcdf.CFNameCoordMap property*), 408
coords (*iris.plot.PlotDefn attribute*), 444
coords() (*iris.cube.Cube method*), 358
CoordSystem (*class in iris.coord_systems*), 315
copy() (*iris.coords.AncillaryVariable method*), 326
copy() (*iris.coords.AuxCoord method*), 330
copy() (*iris.coords.CellMeasure method*), 336
copy() (*iris.coords.Coord method*), 340
copy() (*iris.coords.DimCoord method*), 346
copy() (*iris.cube.Cube method*), 359
copy() (*iris.cube.CubeList method*), 370
copy() (*iris.fileformats.netcdf.OrderedAddableList method*), 409
copy() (*iris.fileformats.pp.PPField method*), 415
core_bounds() (*iris.coords.AuxCoord method*), 330
core_bounds() (*iris.coords.Coord method*), 340
core_bounds() (*iris.coords.DimCoord method*), 347
core_data() (*iris.coords.AncillaryVariable method*), 326
core_data() (*iris.coords.CellMeasure method*), 336
core_data() (*iris.cube.Cube method*), 359
core_data() (*iris.fileformats.pp.PPField method*), 415
core_data() (*iris.fileformats.um.FieldCollation method*), 427
core_points() (*iris.coords.AuxCoord method*), 330
core_points() (*iris.coords.Coord method*), 340
core_points() (*iris.coords.DimCoord method*), 347
cosine_latitude_weights() (*in module iris.analysis.cartography*), 242
COUNT (*in module iris.analysis*), 257
count() (*iris.analysis.cartography.DistanceDifferential method*), 247
count() (*iris.analysis.cartography.PartialDifferential method*), 247
count() (*iris.common.metadata.AncillaryVariableMetadata method*), 289
count() (*iris.common.metadata.BaseMetadata method*), 291
count() (*iris.common.metadata.CellMeasureMetadata method*), 293
count() (*iris.common.metadata.CoordMetadata method*), 294
count() (*iris.common.metadata.CubeMetadata method*), 296
count() (*iris.common.metadata.DimCoordMetadata method*), 298
count() (*iris.coords.Cell method*), 334
count() (*iris.coords.CoordExtent method*), 344
count() (*iris.cube.CubeList method*), 370
count() (*iris.fileformats.name_loaders.NAMECoord method*), 404
count() (*iris.fileformats.netcdf.OrderedAddableList method*), 409
count() (*iris.fileformats.pp.STASH method*), 418
count() (*iris.fileformats.rules.ConversionMetadata method*), 421
count() (*iris.fileformats.rules.Factory method*), 422
count() (*iris.fileformats.rules.Loader method*), 422
count() (*iris.fileformats.rules.ReferenceTarget method*), 423
count() (*iris.fileformats.um_cf_map.CFName method*), 428
count() (*iris.plot.PlotDefn method*), 444
create_temp_filename() (*in module iris.util*), 451
Cube (*class in iris.cube*), 352
cube() (*iris.common.resolve.Resolve method*), 304
cube_delta() (*in module iris.analysis.calculus*), 239
cube_dims() (*iris.coords.AncillaryVariable method*), 326
cube_dims() (*iris.coords.AuxCoord method*), 330
cube_dims() (*iris.coords.CellMeasure method*), 336
cube_dims() (*iris.coords.Coord method*), 340
cube_dims() (*iris.coords.DimCoord method*), 347
cube_text() (*in module iris.fileformats.dot*), 402
CubeList (*class in iris.cube*), 368
CubeListRepresentation (*class in iris.experimental.representation*), 382
CubeMetadata (*class in iris.common.metadata*), 296
CubeRepresentation (*class in iris.experimental.representation*), 383
curl() (*in module iris.analysis.calculus*), 240

D

data (*iris.coords.AncillaryVariable property*), 327
data (*iris.coords.CellMeasure property*), 337
data (*iris.cube.Cube property*), 367
data (*iris.fileformats.pp.PPField property*), 416
data (*iris.fileformats.um.FieldCollation property*), 427
data_field_indices (*iris.fileformats.um.FieldCollation property*), 427
data_filepath (*iris.fileformats.um.FieldCollation property*), 427
data_proxy (*iris.fileformats.um.FieldCollation property*), 427

data_variables (*iris.fileformats.cf.CFGroup property*), 396
day (*iris.time.PartialDateTime attribute*), 448
decode_uri() (*in module iris.io*), 432
DEFAULT_NAME (*iris.common.metadata.AncillaryVariableMetadata attribute*), 290
DEFAULT_NAME (*iris.common.metadata.BaseMetadata attribute*), 292
DEFAULT_NAME (*iris.common.metadata.CellMeasureMetadata attribute*), 294
DEFAULT_NAME (*iris.common.metadata.CoordMetadata attribute*), 296
DEFAULT_NAME (*iris.common.metadata.CubeMetadata attribute*), 298
DEFAULT_NAME (*iris.common.metadata.DimCoordMetadata attribute*), 300
default_projection() (*in module iris.plot*), 440
default_projection_extent() (*in module iris.plot*), 440
delta() (*in module iris.util*), 452
demote_dim_coord_to_aux_coord() (*in module iris.util*), 452
dependencies (*iris.aux_factory.AtmosphereSigmaFactory property*), 274
dependencies (*iris.aux_factory.AuxCoordFactory property*), 275
dependencies (*iris.aux_factory.HybridHeightFactory property*), 277
dependencies (*iris.aux_factory.HybridPressureFactory property*), 279
dependencies (*iris.aux_factory.OceanSFactory property*), 280
dependencies (*iris.aux_factory.OceanSg1Factory property*), 282
dependencies (*iris.aux_factory.OceanSg2Factory property*), 283
dependencies (*iris.aux_factory.OceanSigmaFactory property*), 285
dependencies (*iris.aux_factory.OceanSigmaZFactory property*), 286
deprecated_options (*iris.Future attribute*), 465
derived_coords (*iris.cube.Cube property*), 367
derived_dims() (*iris.aux_factory.AtmosphereSigmaFactory method*), 273
derived_dims() (*iris.aux_factory.AuxCoordFactory method*), 274
derived_dims() (*iris.aux_factory.HybridHeightFactory method*), 276
derived_dims() (*iris.aux_factory.HybridPressureFactory method*), 278
derived_dims() (*iris.aux_factory.OceanSFactory method*), 279
derived_dims() (*iris.aux_factory.OceanSg1Factory method*), 281
derived_dims() (*iris.aux_factory.OceanSg2Factory method*), 282
derived_dims() (*iris.aux_factory.OceanSigmaFactory method*), 284
derived_dims() (*iris.aux_factory.OceanSigmaZFactory method*), 285
describe_diff() (*in module iris.util*), 453
difference() (*iris.common.metadata.AncillaryVariableMetadata method*), 289
difference() (*iris.common.metadata.BaseMetadata method*), 291
difference() (*iris.common.metadata.CellMeasureMetadata method*), 293
difference() (*iris.common.metadata.CoordMetadata method*), 295
difference() (*iris.common.metadata.CubeMetadata method*), 296
difference() (*iris.common.metadata.DimCoordMetadata method*), 299
differentiate() (*in module iris.analysis.calculus*), 239
dim_coords (*iris.cube.Cube property*), 367
dim_coords_and_dims (*iris.fileformats.rules.ConversionMetadata attribute*), 421
DimCoord (*class in iris.coords*), 345
DimCoordMetadata (*class in iris.common.metadata*), 298
dimension (*iris.fileformats.name_loaders.NAMECoord attribute*), 404
DistanceDifferential (*class in iris.analysis.cartography*), 247
divide() (*in module iris.analysis.maths*), 250
documenting.docstrings_attribute module, 217
documenting.docstrings_sample_routine module, 215
dtype (*iris.coords.AncillaryVariable property*), 327
dtype (*iris.coords.AuxCoord property*), 333
dtype (*iris.coords.CellMeasure property*), 337
dtype (*iris.coords.Coord property*), 343
dtype (*iris.coords.DimCoord property*), 350
dtype (*iris.cube.Cube property*), 367
dtype (*iris.fileformats.netcdf.NetCDFDataProxy attribute*), 408
DuplicateDataError (*class in iris.exceptions*), 375
dx1 (*iris.analysis.cartography.DistanceDifferential attribute*), 247
dx1 (*iris.analysis.cartography.PartialDifferential attribute*), 247
dx2 (*iris.analysis.cartography.DistanceDifferential attribute*), 247
dy1 (*iris.analysis.cartography.DistanceDifferential attribute*), 247

dy1 (*iris.analysis.cartography.PartialDifferential attribute*), 247
dy2 (*iris.analysis.cartography.DistanceDifferential attribute*), 247

E

EARTH_RADIUS (*in module iris.fileformats.pp*), 419
element_arrays_and_dims (*iris.fileformats.um.FieldCollation property*), 427
ellipsoid (*iris.coord_systems.AlbersEqualArea attribute*), 315
ellipsoid (*iris.coord_systems.Geostationary attribute*), 317
ellipsoid (*iris.coord_systems.LambertAzimuthalEqualArea attribute*), 318
ellipsoid (*iris.coord_systems.LambertConformal attribute*), 319
ellipsoid (*iris.coord_systems.Mercator attribute*), 320
ellipsoid (*iris.coord_systems.Orthographic attribute*), 321
ellipsoid (*iris.coord_systems.OSGB attribute*), 320
ellipsoid (*iris.coord_systems.RotatedGeogCS attribute*), 322
ellipsoid (*iris.coord_systems.Stereographic attribute*), 323
ellipsoid (*iris.coord_systems.TransverseMercator attribute*), 324
ellipsoid (*iris.coord_systems.VerticalPerspective attribute*), 325
equal() (*iris.common.metadata.AncillaryVariableMetadata method*), 289
equal() (*iris.common.metadata.BaseMetadata method*), 291
equal() (*iris.common.metadata.CellMeasureMetadata method*), 293
equal() (*iris.common.metadata.CoordMetadata method*), 295
equal() (*iris.common.metadata.CubeMetadata method*), 297
equal() (*iris.common.metadata.DimCoordMetadata method*), 299
equalise_attributes() (*in module iris.experimental.equalise_cubes*), 378
equalise_attributes() (*in module iris.util*), 453
ExampleClass (*class in documenting.docstrings_attribute*), 217
exp() (*in module iris.analysis.maths*), 250
expand_filespecs() (*in module iris.io*), 432
exponentiate() (*in module iris.analysis.maths*), 250
extend() (*iris.cube.CubeList method*), 370
extend() (*iris.fileformats.netcdf.OrderedAddableList method*), 409
extract() (*iris.AttributeConstraint method*), 463
extract() (*iris.Constraint method*), 463
extract() (*iris.cube.Cube method*), 359
extract() (*iris.cube.CubeList method*), 370
extract() (*iris.NameConstraint method*), 464
extract_cube() (*iris.cube.CubeList method*), 370
extract_cubes() (*iris.cube.CubeList method*), 370
extract_overlapping() (*iris.cube.CubeList method*), 370

F

factories (*iris.fileformats.rules.ConversionMetadata attribute*), 421
Factory (*class in iris.fileformats.rules*), 422
factory_class (*iris.fileformats.rules.Factory attribute*), 422
false_easting (*iris.coord_systems.AlbersEqualArea attribute*), 315
false_easting (*iris.coord_systems.Geostationary attribute*), 317
false_easting (*iris.coord_systems.LambertAzimuthalEqualArea attribute*), 318
false_easting (*iris.coord_systems.LambertConformal attribute*), 319
false_easting (*iris.coord_systems.Orthographic attribute*), 321
false_easting (*iris.coord_systems.OSGB attribute*), 320
false_easting (*iris.coord_systems.Stereographic attribute*), 323
false_easting (*iris.coord_systems.TransverseMercator attribute*), 324
false_easting (*iris.coord_systems.VerticalPerspective attribute*), 325
false_northing (*iris.coord_systems.AlbersEqualArea attribute*), 315
false_northing (*iris.coord_systems.Geostationary attribute*), 317
false_northing (*iris.coord_systems.LambertAzimuthalEqualArea attribute*), 318
false_northing (*iris.coord_systems.LambertConformal attribute*), 319
false_northing (*iris.coord_systems.Orthographic attribute*), 321
false_northing (*iris.coord_systems.OSGB attribute*), 320
false_northing (*iris.coord_systems.Stereographic attribute*), 323
false_northing (*iris.coord_systems.TransverseMercator attribute*), 324
false_northing (*iris.coord_systems.VerticalPerspective attribute*), 325
field_generator (*iris.fileformats.rules.Loader attribute*), 422

field_generator_kwargs (*iris.fileformats.rules.Loader attribute*), 422
FieldCollation (*class in iris.fileformats.um*), 427
fields (*iris.fileformats.um.FieldCollation property*), 427
file_element (*iris.io.format_picker.FormatSpecification property*), 431
file_element_value (*iris.io.format_picker.FormatSpecification property*), 431
file_is_newer_than() (*in module iris.util*), 453
FileElement (*class in iris.io.format_picker*), 429
FileExtension (*class in iris.io.format_picker*), 429
fill_value (*iris.fileformats.netcdf.NetCDFDataProxy attribute*), 408
find_discontiguities() (*in module iris.util*), 454
find_saver() (*in module iris.io*), 433
FORMAT_AGENT (*in module iris.fileformats*), 428
format_array() (*in module iris.util*), 455
FormatAgent (*class in iris.io.format_picker*), 430
FormatSpecification (*class in iris.io.format_picker*), 430
formula_terms (*iris.fileformats.cf.CFGroup property*), 396
from_coord() (*iris.coords.AuxCoord class method*), 330
from_coord() (*iris.coords.Coord class method*), 340
from_coord() (*iris.coords.DimCoord class method*), 347
from_metadata() (*iris.common.metadata.AncillaryVariableMetadata class method*), 289
from_metadata() (*iris.common.metadata.BaseMetadata class method*), 291
from_metadata() (*iris.common.metadata.CellMeasureMetadata class method*), 293
from_metadata() (*iris.common.metadata.CoordMetadata class method*), 295
from_metadata() (*iris.common.metadata.CubeMetadata class method*), 297
from_metadata() (*iris.common.metadata.DimCoordMetadata class method*), 299
from_msi() (*iris.fileformats.pp.STASH static method*), 418
from_regular() (*iris.coords.DimCoord class method*), 347
Future (*class in iris*), 464
FUTURE (*in module iris*), 465

G

GeogCS (*class in iris.coord_systems*), 316
geometry_area_weights() (*in module iris.analysis.geometry*), 248
Geostationary (*class in iris.coord_systems*), 317
get() (*iris.fileformats.cf.CFGroup method*), 396
get_dir_option() (*in module iris.config*), 309
get_element() (*iris.io.format_picker.FileElement method*), 429
get_element() (*iris.io.format_picker.FileExtension method*), 429
get_element() (*iris.io.format_picker.LeadingLine method*), 431
get_element() (*iris.io.format_picker.MagicNumber method*), 431
get_element() (*iris.io.format_picker.UriProtocol method*), 431
get_logger() (*in module iris.config*), 310
get_option() (*in module iris.config*), 310
get_spec() (*iris.io.format_picker.FormatAgent method*), 430
get_xy_contiguous_bounded_grids() (*in module iris.analysis.cartography*), 242
get_xy_grids() (*in module iris.analysis.cartography*), 242
global_attributes (*iris.fileformats.cf.CFGroup attribute*), 396
GMEAN (*in module iris.analysis*), 257
grid_mapping_name (*iris.coord_systems.AlbersEqualArea attribute*), 315
grid_mapping_name (*iris.coord_systems.CoordSystem attribute*), 315
grid_mapping_name (*iris.coord_systems.GeogCS attribute*), 316
grid_mapping_name (*iris.coord_systems.Geostationary attribute*), 317
grid_mapping_name (*iris.coord_systems.LambertAzimuthalEqualArea attribute*), 318
grid_mapping_name (*iris.coord_systems.LambertConformal attribute*), 319
grid_mapping_name (*iris.coord_systems.Mercator attribute*), 320
grid_mapping_name (*iris.coord_systems.Orthographic attribute*), 321
grid_mapping_name (*iris.coord_systems.OSGB attribute*), 320
grid_mapping_name (*iris.coord_systems.RotatedGeogCS attribute*), 322
grid_mapping_name (*iris.coord_systems.Stereographic attribute*), 323
grid_mapping_name (*iris.coord_systems.TransverseMercator attribute*), 324
grid_mapping_name (*iris.coord_systems.VerticalPerspective attribute*), 325
grid_mappings (*iris.fileformats.cf.CFGroup property*), 397
grid_north_pole_latitude (*iris.coord_systems.RotatedGeogCS attribute*), 322
grid_north_pole_longitude (*iris.coord_systems.RotatedGeogCS attribute*), 322
gridcell_angles() (*in module*

iris.analysis.cartography), 243
guess_bounds() (*iris.coords.AuxCoord method*), 330
guess_bounds() (*iris.coords.Coord method*), 340
guess_bounds() (*iris.coords.DimCoord method*), 347
guess_coord_axis() (*in module iris.util*), 455

H

handler (*iris.io.format_picker.FormatSpecification property*), 431
has_aux_factory() (*in module iris.fileformats.rules*), 420
has_bounds() (*iris.coords.AncillaryVariable method*), 327
has_bounds() (*iris.coords.AuxCoord method*), 331
has_bounds() (*iris.coords.CellMeasure method*), 336
has_bounds() (*iris.coords.Coord method*), 341
has_bounds() (*iris.coords.DimCoord method*), 348
has_formula_terms() (*iris.fileformats.cf.CFAncillaryDataVariable method*), 386
has_formula_terms() (*iris.fileformats.cf.CFAuxiliaryCoordinateVariable method*), 388
has_formula_terms() (*iris.fileformats.cf.CFBoundaryVariable method*), 389
has_formula_terms() (*iris.fileformats.cf.CFClimatologyVariable method*), 391
has_formula_terms() (*iris.fileformats.cf.CFCoordinateVariable method*), 392
has_formula_terms() (*iris.fileformats.cf.CFDataVariable method*), 393
has_formula_terms() (*iris.fileformats.cf.CFGridMappingVariable method*), 395
has_formula_terms() (*iris.fileformats.cf.CFLabelVariable method*), 398
has_formula_terms() (*iris.fileformats.cf.CFMeasureVariable method*), 399
has_formula_terms() (*iris.fileformats.cf.CFVariable method*), 401
has_lazy_bounds() (*iris.coords.AuxCoord method*), 331
has_lazy_bounds() (*iris.coords.Coord method*), 341
has_lazy_bounds() (*iris.coords.DimCoord method*), 348
has_lazy_data() (*iris.coords.AncillaryVariable method*), 327
has_lazy_data() (*iris.coords.CellMeasure method*), 336
has_lazy_data() (*iris.cube.Cube method*), 359
has_lazy_points() (*iris.coords.AuxCoord method*), 331
has_lazy_points() (*iris.coords.Coord method*), 341
has_lazy_points() (*iris.coords.DimCoord method*), 348
hexdigest() (*in module iris.common.metadata*), 300
HMEAN (*in module iris.analysis*), 258
hour (*iris.time.PartialDateTime attribute*), 448
HybridHeightFactory (*class in iris.aux_factory*), 276
HybridPressureFactory (*class in iris.aux_factory*), 277

I

identify() (*iris.fileformats.cf.CFAncillaryDataVariable class method*), 386
identify() (*iris.fileformats.cf.CFAuxiliaryCoordinateVariable class method*), 388
identify() (*iris.fileformats.cf.CFBoundaryVariable class method*), 389
identify() (*iris.fileformats.cf.CFClimatologyVariable class method*), 391
identify() (*iris.fileformats.cf.CFCoordinateVariable class method*), 392
identify() (*iris.fileformats.cf.CFDataVariable class method*), 394
identify() (*iris.fileformats.cf.CFGridMappingVariable class method*), 395
identify() (*iris.fileformats.cf.CFLabelVariable class method*), 398
identify() (*iris.fileformats.cf.CFMeasureVariable class method*), 399
identify() (*iris.fileformats.cf.CFVariable method*), 401
IFunc (*class in iris.analysis.maths*), 252
IgnoreCubeException (*class in iris.exceptions*), 375
index() (*iris.analysis.cartography.DistanceDifferential method*), 247
index() (*iris.analysis.cartography.PartialDifferential method*), 247
index() (*iris.common.metadata.AncillaryVariableMetadata method*), 289
index() (*iris.common.metadata.BaseMetadata method*), 291
index() (*iris.common.metadata.CellMeasureMetadata method*), 293
index() (*iris.common.metadata.CoordMetadata method*), 295
index() (*iris.common.metadata.CubeMetadata method*), 297
index() (*iris.common.metadata.DimCoordMetadata method*), 299
index() (*iris.coords.Cell method*), 334

index() (*iris.coords.CoordExtent method*), 344
index() (*iris.cube.CubeList method*), 371
index() (*iris.fileformats.name_loaders.NAMECoord method*), 404
index() (*iris.fileformats.netcdf.OrderedAddableList method*), 409
index() (*iris.fileformats.pp.STASH method*), 418
index() (*iris.fileformats.rules.ConversionMetadata method*), 421
index() (*iris.fileformats.rules.Factory method*), 422
index() (*iris.fileformats.rules.Loader method*), 422
index() (*iris.fileformats.rules.ReferenceTarget method*), 423
index() (*iris.fileformats.um_cf_map.CFName method*), 428
index() (*iris.plot.PlotDefn method*), 444
insert() (*iris.cube.CubeList method*), 371
insert() (*iris.fileformats.netcdf.OrderedAddableList method*), 409
interpolate() (*in module iris.analysis.trajectory*), 254
interpolate() (*iris.analysis.trajectory.Trajectory method*), 255
interpolate() (*iris.cube.Cube method*), 359
interpolator() (*iris.analysis.Linear method*), 267
interpolator() (*iris.analysis.Nearest method*), 270
intersect() (*iris.coords.AuxCoord method*), 331
intersect() (*iris.coords.Coord method*), 341
intersect() (*iris.coords.DimCoord method*), 348
intersection() (*iris.cube.Cube method*), 360
intersection_of_cubes() (*in module iris.analysis.maths*), 251
intervals (*iris.coords.CellMethod attribute*), 338
InvalidCubeError (*class in iris.exceptions*), 376
inverse() (*iris.palette.SymmetricNormalize method*), 437
inverse_flattening (*iris.coord_systems.GeogCS attribute*), 316
iris
    module, 459
iris.analysis
    module, 256
iris.analysis.calculus
    module, 239
iris.analysis.cartography
    module, 241
iris.analysis.geometry
    module, 248
iris.analysis.maths
    module, 248
iris.analysis.stats
    module, 253
iris.analysis.trajectory
    module, 254
iris.aux_factory
    module, 272
iris.common
    module, 309
iris.common.lenient
    module, 287
iris.common.metadata
    module, 288
iris.common.mixin
    module, 301
iris.common.resolve
    module, 302
iris.config
    module, 309
iris.config.IMPORT_LOGGER (*in module iris.config*), 309
iris.config.PALETTE_PATH (*in module iris.config*), 309
iris.config.TEST_DATA_DIR (*in module iris.config*), 309
iris.coord_categorisation
    module, 311
iris.coord_systems
    module, 314
iris.coords
    module, 325
iris.cube
    module, 351
iris.exceptions
    module, 373
iris.experimental
    module, 384
iris.experimental.animate
    module, 377
iris.experimental.equalise_cubes
    module, 378
iris.experimental.regrid
    module, 378
iris.experimental.regrid_conservative
    module, 382
iris.experimental.representation
    module, 382
iris.experimental.stratify
    module, 383
iris.experimental.ugrid
    module, 384
iris.fileformats
    module, 428
iris.fileformats.abf
    module, 384
iris.fileformats.cf
    module, 385
iris.fileformats.dot
    module, 402
iris.fileformats.name

module, 402
iris.fileformats.name_loaders
module, 403
iris.fileformats.netcdf
module, 405
iris.fileformats.nimrod
module, 412
iris.fileformats.nimrod_load_rules
module, 413
iris.fileformats.pp
module, 413
iris.fileformats.pp_load_rules
module, 419
iris.fileformats.pp_save_rules
module, 419
iris.fileformats.rules
module, 419
iris.fileformats.um
module, 423
iris.fileformats.um_cf_map
module, 428
iris.io
module, 431
iris.io.format_picker
module, 428
iris.iterate
module, 435
iris.palette
module, 435
iris.pandas
module, 437
iris.plot
module, 438
iris.quickplot
module, 444
iris.std_names
module, 446
iris.symbols
module, 447
iris.time
module, 447
iris.util
module, 448
IrisDeprecation (*class in iris*), 465
IrisError (*class in iris.exceptions*), 376
is_brewer() (*in module iris.palette*), 436
is_compatible() (*iris.coords.AncillaryVariable method*), 327
is_compatible() (*iris.coords.AuxCoord method*), 331
is_compatible() (*iris.coords.CellMeasure method*), 336
is_compatible() (*iris.coords.Coord method*), 341
is_compatible() (*iris.coords.DimCoord method*), 348
is_compatible() (*iris.cube.Cube method*), 361
is_contiguous() (*iris.coords.AuxCoord method*), 331
is_contiguous() (*iris.coords.Coord method*), 341
is_contiguous() (*iris.coords.DimCoord method*), 348
is_monotonic() (*iris.coords.AuxCoord method*), 331
is_monotonic() (*iris.coords.Coord method*), 341
is_monotonic() (*iris.coords.DimCoord method*), 349
is_regular() (*in module iris.util*), 455
is_valid (*iris.fileformats.pp.STASH property*), 419
item (*iris.fileformats.pp.STASH attribute*), 419
items() (*iris.fileformats.cf.CFGroup method*), 396
izip() (*in module iris.iterate*), 435

K

keys() (*iris.fileformats.cf.CFGroup method*), 396

L

labels (*iris.fileformats.cf.CFGroup property*), 397
LambertAzimuthalEqualArea (*class in iris.coord_systems*), 318
LambertConformal (*class in iris.coord_systems*), 319
latitude_of_projection_origin (*iris.coord_systems.AlbersEqualArea attribute*), 315
latitude_of_projection_origin (*iris.coord_systems.Geostationary attribute*), 317
latitude_of_projection_origin (*iris.coord_systems.LambertAzimuthalEqualArea attribute*), 318
latitude_of_projection_origin (*iris.coord_systems.Orthographic attribute*), 321
latitude_of_projection_origin (*iris.coord_systems.OSGB attribute*), 320
latitude_of_projection_origin (*iris.coord_systems.TransverseMercator attribute*), 324
latitude_of_projection_origin (*iris.coord_systems.VerticalPerspective attribute*), 325
lazy_aggregate() (*iris.analysis.Aggregator method*), 264
lazy_aggregate() (*iris.analysis.WeightedAggregator method*), 266
lazy_bounds() (*iris.coords.AuxCoord method*), 331
lazy_bounds() (*iris.coords.Coord method*), 341
lazy_bounds() (*iris.coords.DimCoord method*), 349
lazy_data() (*iris.coords.AncillaryVariable method*), 327
lazy_data() (*iris.coords.CellMeasure method*), 336
lazy_data() (*iris.cube.Cube method*), 362
lazy_func (*iris.analysis.Aggregator attribute*), 265
lazy_func (*iris.analysis.WeightedAggregator attribute*), 267

lazy_points() (*iris.coords.AuxCoord method*), 332
lazy_points() (*iris.coords.Coord method*), 342
lazy_points() (*iris.coords.DimCoord method*), 349
LazyAggregatorError (*class in iris.exceptions*), 376
lbcode (*iris.fileformats.pp.PPField property*), 416
lbpack (*iris.fileformats.pp.PPField property*), 416
lbproc (*iris.fileformats.pp.PPField property*), 416
lbtim (*iris.fileformats.pp.PPField property*), 416
lbuser3() (*iris.fileformats.pp.STASH method*), 418
lbuser6() (*iris.fileformats.pp.STASH method*), 419
LeadingLine (*class in iris.io.format_picker*), 431
len_formats (*iris.io.format_picker.MagicNumber attribute*), 431
Lenient (*class in iris.common.lenient*), 287
LENIENT (*in module iris.common.lenient*), 287
lhs_cube (*iris.common.resolve.Resolve attribute*), 305
lhs_cube_aux_coverage (*iris.common.resolve.Resolve attribute*), 305
lhs_cube_category (*iris.common.resolve.Resolve attribute*), 305
lhs_cube_category_local (*iris.common.resolve.Resolve attribute*), 305
lhs_cube_dim_coverage (*iris.common.resolve.Resolve attribute*), 305
lhs_cube_resolved (*iris.common.resolve.Resolve attribute*), 306
Linear (*class in iris.analysis*), 267
LINEAR_EXTRAPOLATION_MODES (*iris.analysis.Linear attribute*), 268
load() (*in module iris*), 460
load() (*in module iris.fileformats.pp*), 413
load_cube() (*in module iris*), 460
load_cubes() (*in module iris*), 461
load_cubes() (*in module iris.fileformats.abf*), 385
load_cubes() (*in module iris.fileformats.name*), 402
load_cubes() (*in module iris.fileformats.netcdf*), 405
load_cubes() (*in module iris.fileformats.nimrod*), 412
load_cubes() (*in module iris.fileformats.pp*), 414
load_cubes() (*in module iris.fileformats.rules*), 420
load_cubes() (*in module iris.fileformats.um*), 424
load_cubes_32bit_ieee() (*in module iris.fileformats.um*), 424
load_files() (*in module iris.io*), 433
load_http() (*in module iris.io*), 433
load_NAMEII_field() (*in module iris.fileformats.name_loaders*), 404
load_NAMEII_timeseries() (*in module iris.fileformats.name_loaders*), 404
load_NAMEIII_field() (*in module iris.fileformats.name_loaders*), 403
load_NAMEIII_timeseries() (*in module iris.fileformats.name_loaders*), 403
load_NAMEIII_trajectory() (*in module iris.fileformats.name_loaders*), 403
load_NAMEIII_version2() (*in module iris.fileformats.name_loaders*), 403
load_pairs_from_fields() (*in module iris.fileformats.pp*), 416
load_pairs_from_fields() (*in module iris.fileformats.rules*), 420
load_raw() (*in module iris*), 461
Loader (*class in iris.fileformats.rules*), 422
log() (*in module iris.analysis.maths*), 251
log10() (*in module iris.analysis.maths*), 251
log2() (*in module iris.analysis.maths*), 251
long_name (*iris.aux_factory.AtmosphereSigmaFactory property*), 274
long_name (*iris.aux_factory.AuxCoordFactory property*), 275
long_name (*iris.aux_factory.HybridHeightFactory property*), 277
long_name (*iris.aux_factory.HybridPressureFactory property*), 279
long_name (*iris.aux_factory.OceanSFactory property*), 280
long_name (*iris.aux_factory.OceanSg1Factory property*), 282
long_name (*iris.aux_factory.OceanSg2Factory property*), 283
long_name (*iris.aux_factory.OceanSigmaFactory property*), 285
long_name (*iris.aux_factory.OceanSigmaZFactory property*), 286
long_name (*iris.common.metadata.AncillaryVariableMetadata attribute*), 290
long_name (*iris.common.metadata.BaseMetadata attribute*), 292
long_name (*iris.common.metadata.CellMeasureMetadata attribute*), 294
long_name (*iris.common.metadata.CoordMetadata attribute*), 296
long_name (*iris.common.metadata.CubeMetadata attribute*), 298
long_name (*iris.common.metadata.DimCoordMetadata attribute*), 300
long_name (*iris.common.mixin.CFVariableMixin property*), 301
long_name (*iris.coords.AncillaryVariable property*), 328
long_name (*iris.coords.AuxCoord property*), 333
long_name (*iris.coords.CellMeasure property*), 337
long_name (*iris.coords.Coord property*), 343
long_name (*iris.coords.DimCoord property*), 350
long_name (*iris.cube.Cube property*), 367
long_name (*iris.fileformats.rules.ConversionMetadata attribute*), 421
long_name (*iris.fileformats.um_cf_map.CFName attribute*), 428
longitude_of_central_meridian

(*iris.coord_systems.AlbersEqualArea attribute*), 315
longitude_of_central_meridian (*iris.coord_systems.OSGB attribute*), 320
longitude_of_central_meridian (*iris.coord_systems.TransverseMercator attribute*), 324
longitude_of_prime_meridian (*iris.coord_systems.GeogCS attribute*), 316
longitude_of_projection_origin (*iris.coord_systems.Geostationary attribute*), 317
longitude_of_projection_origin (*iris.coord_systems.LambertAzimuthalEqualArea attribute*), 318
longitude_of_projection_origin (*iris.coord_systems.Mercator attribute*), 320
longitude_of_projection_origin (*iris.coord_systems.Orthographic attribute*), 321
longitude_of_projection_origin (*iris.coord_systems.VerticalPerspective attribute*), 325

M

MagicNumber (*class in iris.io.format_picker*), 431
make_content() (*iris.experimental.representation.CubeListRepresentation method*), 382
make_coord() (*iris.aux_factory.AtmosphereSigmaFactory method*), 273
make_coord() (*iris.aux_factory.AuxCoordFactory method*), 274
make_coord() (*iris.aux_factory.HybridHeightFactory method*), 276
make_coord() (*iris.aux_factory.HybridPressureFactory method*), 278
make_coord() (*iris.aux_factory.OceanSFactory method*), 279
make_coord() (*iris.aux_factory.OceanSg1Factory method*), 281
make_coord() (*iris.aux_factory.OceanSg2Factory method*), 282
make_coord() (*iris.aux_factory.OceanSigmaFactory method*), 284
make_coord() (*iris.aux_factory.OceanSigmaZFactory method*), 285
map_rhs_to_lhs (*iris.common.resolve.Resolve attribute*), 306
mapped (*iris.common.resolve.Resolve property*), 306
mapping (*iris.common.resolve.Resolve attribute*), 307
mask_cube() (*in module iris.util*), 455
MAX (*in module iris.analysis*), 258
max_inclusive (*iris.coords.CoordExtent attribute*), 344
maximum (*iris.coords.CoordExtent attribute*), 344
MEAN (*in module iris.analysis*), 258
measure (*iris.common.metadata.CellMeasureMetadata attribute*), 294
measure (*iris.coords.CellMeasure property*), 337
MEDIAN (*in module iris.analysis*), 259
Mercator (*class in iris.coord_systems*), 319
merge() (*iris.cube.CubeList method*), 371
merge_cube() (*iris.cube.CubeList method*), 372
MergeError (*class in iris.exceptions*), 376
metadata (*iris.aux_factory.AtmosphereSigmaFactory property*), 274
metadata (*iris.aux_factory.AuxCoordFactory property*), 275
metadata (*iris.aux_factory.HybridHeightFactory property*), 277
metadata (*iris.aux_factory.HybridPressureFactory property*), 279
metadata (*iris.aux_factory.OceanSFactory property*), 280
metadata (*iris.aux_factory.OceanSg1Factory property*), 282
metadata (*iris.aux_factory.OceanSg2Factory property*), 283
metadata (*iris.aux_factory.OceanSigmaFactory property*), 285
metadata (*iris.aux_factory.OceanSigmaZFactory property*), 287
metadata (*iris.common.mixin.CFVariableMixin property*), 301
metadata (*iris.coords.AncillaryVariable property*), 328
metadata (*iris.coords.AuxCoord property*), 333
metadata (*iris.coords.CellMeasure property*), 337
metadata (*iris.coords.Coord property*), 343
metadata (*iris.coords.DimCoord property*), 350
metadata (*iris.cube.Cube property*), 368
metadata_manager_factory() (*in module iris.common.metadata*), 300
method (*iris.coords.CellMethod attribute*), 338
microsecond (*iris.time.PartialDateTime attribute*), 448
MIN (*in module iris.analysis*), 259
min_inclusive (*iris.coords.CoordExtent attribute*), 344
minimum (*iris.coords.CoordExtent attribute*), 345
minute (*iris.time.PartialDateTime attribute*), 448
model (*iris.fileformats.pp.STASH attribute*), 419
module
    documenting.docstrings_attribute, 217
    documenting.docstrings_sample_routine, 215
    iris, 459
    iris.analysis, 256
    iris.analysis.calculus, 239
    iris.analysis.cartography, 241
    iris.analysis.geometry, 248

iris.analysis.maths, 248
iris.analysis.stats, 253
iris.analysis.trajectory, 254
iris.aux_factory, 272
iris.common, 309
iris.common.lenient, 287
iris.common.metadata, 288
iris.common.mixin, 301
iris.common.resolve, 302
iris.config, 309
iris.coord_categorisation, 311
iris.coord_systems, 314
iris.coords, 325
iris.cube, 351
iris.exceptions, 373
iris.experimental, 384
iris.experimental.animate, 377
iris.experimental.equalise_cubes, 378
iris.experimental.regrid, 378
iris.experimental.regrid_conservative, 382
iris.experimental.representation, 382
iris.experimental.stratify, 383
iris.experimental.ugrid, 384
iris.fileformats, 428
iris.fileformats.abf, 384
iris.fileformats.cf, 385
iris.fileformats.dot, 402
iris.fileformats.name, 402
iris.fileformats.name_loaders, 403
iris.fileformats.netcdf, 405
iris.fileformats.nimrod, 412
iris.fileformats.nimrod_load_rules, 413
iris.fileformats.pp, 413
iris.fileformats.pp_load_rules, 419
iris.fileformats.pp_save_rules, 419
iris.fileformats.rules, 419
iris.fileformats.um, 423
iris.fileformats.um_cf_map, 428
iris.io, 431
iris.io.format_picker, 428
iris.iterate, 435
iris.palette, 435
iris.pandas, 437
iris.plot, 438
iris.quickplot, 444
iris.std_names, 446
iris.symbols, 447
iris.time, 447
iris.util, 448
monotonic() (*in module iris.util*), 455
month (*iris.time.PartialDateTime attribute*), 448
multiply() (*in module iris.analysis.maths*), 251

N

name (*iris.fileformats.name_loaders.NAMECoord attribute*), 404
name (*iris.fileformats.rules.ConcreteReferenceTarget attribute*), 421
name (*iris.fileformats.rules.ReferenceTarget attribute*), 423
name (*iris.io.format_picker.FormatSpecification property*), 431
name() (*iris.analysis.Aggregator method*), 264
name() (*iris.analysis.WeightedAggregator method*), 266
name() (*iris.aux_factory.AtmosphereSigmaFactory method*), 273
name() (*iris.aux_factory.AuxCoordFactory method*), 275
name() (*iris.aux_factory.HybridHeightFactory method*), 276
name() (*iris.aux_factory.HybridPressureFactory method*), 278
name() (*iris.aux_factory.OceanSFactory method*), 279
name() (*iris.aux_factory.OceanSg1Factory method*), 281
name() (*iris.aux_factory.OceanSg2Factory method*), 282
name() (*iris.aux_factory.OceanSigmaFactory method*), 284
name() (*iris.aux_factory.OceanSigmaZFactory method*), 286
name() (*iris.common.metadata.AncillaryVariableMetadata method*), 289
name() (*iris.common.metadata.BaseMetadata method*), 291
name() (*iris.common.metadata.CellMeasureMetadata method*), 293
name() (*iris.common.metadata.CoordMetadata method*), 295
name() (*iris.common.metadata.CubeMetadata method*), 297
name() (*iris.common.metadata.DimCoordMetadata method*), 299
name() (*iris.common.mixin.CFVariableMixin method*), 301
name() (*iris.coords.AncillaryVariable method*), 327
name() (*iris.coords.AuxCoord method*), 332
name() (*iris.coords.CellMeasure method*), 336
name() (*iris.coords.Coord method*), 342
name() (*iris.coords.DimCoord method*), 349
name() (*iris.cube.Cube method*), 362
name() (*iris.fileformats.netcdf.CFNameCoordMap method*), 408
name_or_coord (*iris.coords.CoordExtent attribute*), 345
NameConstraint (*class in iris*), 464
NAMECoord (*class in iris.fileformats.name_loaders*), 404
names (*iris.fileformats.netcdf.CFNameCoordMap property*), 408
nbounds (*iris.coords.AuxCoord property*), 333
nbounds (*iris.coords.Coord property*), 343

nbounds (*iris.coords.DimCoord property*), 350
ndim (*iris.coords.AncillaryVariable property*), 328
ndim (*iris.coords.AuxCoord property*), 333
ndim (*iris.coords.CellMeasure property*), 337
ndim (*iris.coords.Coord property*), 343
ndim (*iris.coords.DimCoord property*), 351
ndim (*iris.cube.Cube property*), 368
ndim (*iris.fileformats.netcdf.NetCDFDataProxy property*), 408
Nearest (*class in iris.analysis*), 269
nearest_neighbour_index() (*iris.coords.AuxCoord method*), 332
nearest_neighbour_index() (*iris.coords.Coord method*), 342
nearest_neighbour_index() (*iris.coords.DimCoord method*), 349
NetCDF (*class in iris.config*), 310
netcdf (*in module iris.config*), 309
NetCDFDataProxy (*class in iris.fileformats.netcdf*), 408
new_axis() (*in module iris.util*), 456
NimrodField (*class in iris.fileformats.nimrod*), 412
north_pole_grid_longitude (*iris.coord_systems.RotatedGeogCS attribute*), 322
NotYetImplementedError (*class in iris.exceptions*), 376

O

OceanSFactory (*class in iris.aux_factory*), 279
OceanSg1Factory (*class in iris.aux_factory*), 281
OceanSg2Factory (*class in iris.aux_factory*), 282
OceanSigmaFactory (*class in iris.aux_factory*), 284
OceanSigmaZFactory (*class in iris.aux_factory*), 285
OrderedAddableList (*class in iris.fileformats.netcdf*), 408
orography_at_bounds() (*in module iris.plot*), 441
orography_at_points() (*in module iris.plot*), 441
Orthographic (*class in iris.coord_systems*), 321
OSGB (*class in iris.coord_systems*), 320
outline() (*in module iris.plot*), 441
outline() (*in module iris.quickplot*), 445

P

parse_cell_methods() (*in module iris.fileformats.netcdf*), 405
PartialDateTime (*class in iris.time*), 447
PartialDifferential (*class in iris.analysis.cartography*), 247
path (*iris.fileformats.netcdf.NetCDFDataProxy attribute*), 408
pcolor() (*in module iris.plot*), 441
pcolor() (*in module iris.quickplot*), 446
pcolormesh() (*in module iris.plot*), 442
pcolormesh() (*in module iris.quickplot*), 446
PEAK (*in module iris.analysis*), 259
pearsonr() (*in module iris.analysis.stats*), 253
PERCENTILE (*in module iris.analysis*), 260
perspective_point_height (*iris.coord_systems.Geostationary attribute*), 317
perspective_point_height (*iris.coord_systems.VerticalPerspective attribute*), 325
plot() (*in module iris.plot*), 442
plot() (*in module iris.quickplot*), 446
PlotDefn (*class in iris.plot*), 444
point (*iris.coords.Cell attribute*), 335
PointInCell (*class in iris.analysis*), 272
PointInCell (*class in iris.experimental.regrid*), 380
points (*iris.coords.AuxCoord property*), 333
points (*iris.coords.Coord property*), 343
points (*iris.coords.DimCoord property*), 351
points() (*in module iris.plot*), 442
points() (*in module iris.quickplot*), 446
points_step() (*in module iris.util*), 456
pop() (*iris.cube.CubeList method*), 372
pop() (*iris.fileformats.cf.CFGroup method*), 396
pop() (*iris.fileformats.netcdf.OrderedAddableList method*), 409
popitem() (*iris.fileformats.cf.CFGroup method*), 396
post_process() (*iris.analysis.Aggregator method*), 264
post_process() (*iris.analysis.WeightedAggregator method*), 266
PPField (*class in iris.fileformats.pp*), 414
prepared_category (*iris.common.resolve.Resolve attribute*), 307
prepared_factories (*iris.common.resolve.Resolve attribute*), 307
process_value() (*iris.palette.SymmetricNormalize static method*), 437
project() (*in module iris.analysis.cartography*), 243
ProjectedUnstructuredLinear (*class in iris.experimental.regrid*), 380
ProjectedUnstructuredNearest (*class in iris.experimental.regrid*), 381
promote_aux_coord_to_dim_coord() (*in module iris.util*), 456
promoted (*iris.fileformats.cf.CFGroup attribute*), 397
PROPORTION (*in module iris.analysis*), 260
Python Enhancement Proposals
    PEP 224, 214
    PEP 257, 214
    PEP 517, 477
    PEP 518, 477

Q

quiver() (*in module iris.plot*), 443

R

read() (*iris.fileformats.nimrod.NimrodField method*), 412
read_header() (*in module iris.fileformats.name_loaders*), 404
realise_data() (*iris.cube.CubeList method*), 372
realised_dtype (*iris.fileformats.um.FieldCollation property*), 427
Reference (*class in iris.fileformats.rules*), 423
references (*iris.fileformats.rules.ConversionMetadata attribute*), 421
ReferenceTarget (*class in iris.fileformats.rules*), 423
regrid() (*iris.cube.Cube method*), 362
regrid_area_weighted_rectilinear_src_and_grid() (*in module iris.experimental.regrid*), 378
regrid_conservative_via_esmpy() (*in module iris.experimental.regrid_conservative*), 382
regrid_weighted_curvilinear_to_rectilinear() (*in module iris.experimental.regrid*), 379
regridder() (*iris.analysis.AreaWeighted method*), 269
regridder() (*iris.analysis.Linear method*), 268
regridder() (*iris.analysis.Nearest method*), 270
regridder() (*iris.analysis.PointInCell method*), 272
regridder() (*iris.analysis.UnstructuredNearest method*), 271
regridder() (*iris.experimental.regrid.ProjectedUnstructuredLinear method*), 380
regridder() (*iris.experimental.regrid.ProjectedUnstructuredNearest method*), 381
regular_step() (*in module iris.util*), 457
relevel() (*in module iris.experimental.stratify*), 383
remove() (*iris.cube.CubeList method*), 373
remove() (*iris.fileformats.netcdf.OrderedAddableList method*), 409
remove_ancillary_variable() (*iris.cube.Cube method*), 363
remove_aux_factory() (*iris.cube.Cube method*), 363
remove_cell_measure() (*iris.cube.Cube method*), 363
remove_coord() (*iris.cube.Cube method*), 363
rename() (*iris.aux_factory.AtmosphereSigmaFactory method*), 273
rename() (*iris.aux_factory.AuxCoordFactory method*), 275
rename() (*iris.aux_factory.HybridHeightFactory method*), 276
rename() (*iris.aux_factory.HybridPressureFactory method*), 278
rename() (*iris.aux_factory.OceanSFactory method*), 280
rename() (*iris.aux_factory.OceanSg1Factory method*), 281
rename() (*iris.aux_factory.OceanSg2Factory method*), 283
rename() (*iris.aux_factory.OceanSigmaFactory method*), 284
rename() (*iris.aux_factory.OceanSigmaZFactory method*), 286
rename() (*iris.common.mixin.CFVariableMixin method*), 301
rename() (*iris.coords.AncillaryVariable method*), 327
rename() (*iris.coords.AuxCoord method*), 332
rename() (*iris.coords.CellMeasure method*), 336
rename() (*iris.coords.Coord method*), 342
rename() (*iris.coords.DimCoord method*), 350
rename() (*iris.cube.Cube method*), 363
replace_coord() (*iris.cube.Cube method*), 363
repr_html() (*iris.experimental.representation.CubeListRepresentation method*), 382
repr_html() (*iris.experimental.representation.CubeRepresentation method*), 383
Resolve (*class in iris.common.resolve*), 303
reverse() (*in module iris.util*), 457
reverse() (*iris.cube.CubeList method*), 373
reverse() (*iris.fileformats.netcdf.OrderedAddableList method*), 409
rhs_cube (*iris.common.resolve.Resolve attribute*), 307
rhs_cube_aux_coverage (*iris.common.resolve.Resolve attribute*), 307
rhs_cube_category (*iris.common.resolve.Resolve attribute*), 307
rhs_cube_category_local (*iris.common.resolve.Resolve attribute*), 307
rhs_cube_dim_coverage (*iris.common.resolve.Resolve attribute*), 308
rhs_cube_resolved (*iris.common.resolve.Resolve attribute*), 308
RMS (*in module iris.analysis*), 261
rolling_window() (*in module iris.util*), 458
rolling_window() (*iris.cube.Cube method*), 364
rotate_grid_vectors() (*in module iris.analysis.cartography*), 244
rotate_pole() (*in module iris.analysis.cartography*), 245
rotate_winds() (*in module iris.analysis.cartography*), 245
RotatedGeogCS (*class in iris.coord_systems*), 322
run() (*in module iris.fileformats.nimrod_load_rules*), 413
run_callback() (*in module iris.io*), 433

S

sample_data_path() (*in module iris*), 464
sample_routine() (*in module documenting.docstrings_sample_routine*), 215
sampled_points (*iris.analysis.trajectory.Trajectory attribute*), 255
save() (*in module iris*), 461
save() (*in module iris.fileformats.dot*), 402
save() (*in module iris.fileformats.netcdf*), 405

save() (*in module iris.fileformats.pp*), 414
save() (*in module iris.io*), 433
save() (*iris.fileformats.pp.PPField method*), 415
save_fields() (*in module iris.fileformats.pp*), 417
save_pairs_from_cube() (*in module iris.fileformats.pp*), 417
save_png() (*in module iris.fileformats.dot*), 402
Saver (*class in iris.fileformats.netcdf*), 409
scalar_cell_method() (*in module iris.fileformats.rules*), 420
scalar_coord() (*in module iris.fileformats.rules*), 420
scale_factor_at_central_meridian (*iris.coord_systems.OSGB attribute*), 321
scale_factor_at_central_meridian (*iris.coord_systems.TransverseMercator attribute*), 324
scaled() (*iris.palette.SymmetricNormalize method*), 437
scatter() (*in module iris.plot*), 443
scatter() (*in module iris.quickplot*), 446
secant_latitudes (*iris.coord_systems.LambertConformal attribute*), 319
second (*iris.time.PartialDateTime attribute*), 448
section (*iris.fileformats.pp.STASH attribute*), 419
semi_major_axis (*iris.coord_systems.GeogCS attribute*), 316
semi_minor_axis (*iris.coord_systems.GeogCS attribute*), 316
SERVICES (*in module iris.common.metadata*), 288
SERVICES_COMBINE (*in module iris.common.metadata*), 288
SERVICES_DIFFERENCE (*in module iris.common.metadata*), 288
SERVICES_EQUAL (*in module iris.common.metadata*), 288
setdefault() (*iris.fileformats.cf.CFGroup method*), 396
shape (*iris.common.resolve.Resolve property*), 308
shape (*iris.coords.AncillaryVariable property*), 328
shape (*iris.coords.AuxCoord property*), 333
shape (*iris.coords.CellMeasure property*), 337
shape (*iris.coords.Coord property*), 343
shape (*iris.coords.DimCoord property*), 351
shape (*iris.cube.Cube property*), 368
shape (*iris.fileformats.netcdf.NetCDFDataProxy attribute*), 408
site_configuration (*in module iris*), 464
slices() (*iris.cube.Cube method*), 365
slices_over() (*iris.cube.Cube method*), 365
sort() (*iris.cube.CubeList method*), 373
sort() (*iris.fileformats.netcdf.OrderedAddableList method*), 409
spans() (*iris.fileformats.cf.CFAncillaryDataVariable method*), 386
spans() (*iris.fileformats.cf.CFAuxiliaryCoordinateVariable method*), 388
spans() (*iris.fileformats.cf.CFBoundaryVariable method*), 389
spans() (*iris.fileformats.cf.CFClimatologyVariable method*), 391
spans() (*iris.fileformats.cf.CFCoordinateVariable method*), 392
spans() (*iris.fileformats.cf.CFDataVariable method*), 394
spans() (*iris.fileformats.cf.CFGridMappingVariable method*), 395
spans() (*iris.fileformats.cf.CFLabelVariable method*), 398
spans() (*iris.fileformats.cf.CFMeasureVariable method*), 399
spans() (*iris.fileformats.cf.CFVariable method*), 401
square (*documenting.docstrings_attribute.ExampleClass property*), 217
squeeze() (*in module iris.util*), 458
standard_name (*iris.aux_factory.AtmosphereSigmaFactory property*), 274
standard_name (*iris.aux_factory.AuxCoordFactory property*), 275
standard_name (*iris.aux_factory.HybridHeightFactory property*), 277
standard_name (*iris.aux_factory.HybridPressureFactory property*), 279
standard_name (*iris.aux_factory.OceanSFactory property*), 280
standard_name (*iris.aux_factory.OceanSg1Factory property*), 282
standard_name (*iris.aux_factory.OceanSg2Factory property*), 283
standard_name (*iris.aux_factory.OceanSigmaFactory property*), 285
standard_name (*iris.aux_factory.OceanSigmaZFactory property*), 287
standard_name (*iris.common.metadata.AncillaryVariableMetadata attribute*), 290
standard_name (*iris.common.metadata.BaseMetadata attribute*), 292
standard_name (*iris.common.metadata.CellMeasureMetadata attribute*), 294
standard_name (*iris.common.metadata.CoordMetadata attribute*), 296
standard_name (*iris.common.metadata.CubeMetadata attribute*), 298
standard_name (*iris.common.metadata.DimCoordMetadata attribute*), 300
standard_name (*iris.common.mixin.CFVariableMixin property*), 301
standard_name (*iris.coords.AncillaryVariable property*), 328

standard_name (*iris.coords.AuxCoord property*), 333
standard_name (*iris.coords.CellMeasure property*), 337
standard_name (*iris.coords.Coord property*), 343
standard_name (*iris.coords.DimCoord property*), 351
standard_name (*iris.cube.Cube property*), 368
standard_name (*iris.fileformats.rules.ConversionMetadata attribute*), 421
standard_name (*iris.fileformats.um_cf_map.CFName attribute*), 428
standard_parallel (*iris.coord_systems.Mercator attribute*), 320
standard_parallels (*iris.coord_systems.AlbersEqualArea attribute*), 315
STASH (*class in iris.fileformats.pp*), 418
stash (*iris.fileformats.pp.PPField property*), 416
STD_DEV (*in module iris.analysis*), 261
Stereographic (*class in iris.coord_systems*), 322
structured_um_loading() (*in module iris.fileformats.um*), 424
subset() (*iris.cube.Cube method*), 366
subtract() (*in module iris.analysis.maths*), 252
SUM (*in module iris.analysis*), 261
summary() (*iris.cube.Cube method*), 366
sweep_angle_axis (*iris.coord_systems.Geostationary attribute*), 317
symbols() (*in module iris.plot*), 444
SymmetricNormalize (*class in iris.palette*), 436

T

t1 (*iris.fileformats.pp.PPField property*), 416
t2 (*iris.fileformats.pp.PPField property*), 416
time_unit() (*iris.fileformats.pp.PPField method*), 416
timetuple (*iris.time.PartialDateTime attribute*), 448
to_cube() (*iris.fileformats.abf.ABFField method*), 385
token() (*iris.common.metadata.AncillaryVariableMetadata class method*), 290
token() (*iris.common.metadata.BaseMetadata class method*), 292
token() (*iris.common.metadata.CellMeasureMetadata class method*), 294
token() (*iris.common.metadata.CoordMetadata class method*), 295
token() (*iris.common.metadata.CubeMetadata class method*), 297
token() (*iris.common.metadata.DimCoordMetadata class method*), 299
Trajectory (*class in iris.analysis.trajectory*), 255
transform (*iris.fileformats.rules.ConcreteReferenceTarget attribute*), 421
transform (*iris.fileformats.rules.ReferenceTarget attribute*), 423
TranslationError (*class in iris.exceptions*), 377
transpose (*iris.plot.PlotDefn attribute*), 444
transpose() (*iris.cube.Cube method*), 366
TransverseMercator (*class in iris.coord_systems*), 323
true_scale_lat (*iris.coord_systems.Stereographic attribute*), 323

U

ugrid() (*in module iris.experimental.ugrid*), 384
um_to_pp() (*in module iris.fileformats.um*), 423
unify_time_units() (*in module iris.util*), 459
UnitConversionError (*class in iris.exceptions*), 377
units (*iris.aux_factory.AtmosphereSigmaFactory property*), 274
units (*iris.aux_factory.AuxCoordFactory property*), 276
units (*iris.aux_factory.HybridHeightFactory property*), 277
units (*iris.aux_factory.HybridPressureFactory property*), 279
units (*iris.aux_factory.OceanSFactory property*), 280
units (*iris.aux_factory.OceanSg1Factory property*), 282
units (*iris.aux_factory.OceanSg2Factory property*), 283
units (*iris.aux_factory.OceanSigmaFactory property*), 285
units (*iris.aux_factory.OceanSigmaZFactory property*), 287
units (*iris.common.metadata.AncillaryVariableMetadata attribute*), 290
units (*iris.common.metadata.BaseMetadata attribute*), 292
units (*iris.common.metadata.CellMeasureMetadata attribute*), 294
units (*iris.common.metadata.CoordMetadata attribute*), 296
units (*iris.common.metadata.CubeMetadata attribute*), 298
units (*iris.common.metadata.DimCoordMetadata attribute*), 300
units (*iris.common.mixin.CFVariableMixin property*), 301
units (*iris.coords.AncillaryVariable property*), 328
units (*iris.coords.AuxCoord property*), 334
units (*iris.coords.CellMeasure property*), 337
units (*iris.coords.Coord property*), 344
units (*iris.coords.DimCoord property*), 351
units (*iris.cube.Cube property*), 368
units (*iris.fileformats.rules.ConversionMetadata attribute*), 421
units (*iris.fileformats.um_cf_map.CFName attribute*), 428
units_func (*iris.analysis.Aggregator attribute*), 265
units_func (*iris.analysis.WeightedAggregator attribute*), 267
UnknownCellMethodWarning (*class in iris.fileformats.netcdf*), 412
unrotate_pole() (*in module iris.analysis.cartography*), 246

UnstructuredNearest (*class in iris.analysis*), 271
UnstructuredNearestNeigbourRegridder (*class in iris.analysis.trajectory*), 255
update() (*iris.aux_factory.AtmosphereSigmaFactory method*), 273
update() (*iris.aux_factory.AuxCoordFactory method*), 275
update() (*iris.aux_factory.HybridHeightFactory method*), 277
update() (*iris.aux_factory.HybridPressureFactory method*), 278
update() (*iris.aux_factory.OceanSFactory method*), 280
update() (*iris.aux_factory.OceanSg1Factory method*), 281
update() (*iris.aux_factory.OceanSg2Factory method*), 283
update() (*iris.aux_factory.OceanSigmaFactory method*), 284
update() (*iris.aux_factory.OceanSigmaZFactory method*), 286
update() (*iris.fileformats.cf.CFGroup method*), 396
update_global_attributes() (*iris.fileformats.netcdf.Saver method*), 410
update_metadata() (*iris.analysis.Aggregator method*), 264
update_metadata() (*iris.analysis.WeightedAggregator method*), 266
updated() (*iris.aux_factory.AtmosphereSigmaFactory method*), 273
updated() (*iris.aux_factory.AuxCoordFactory method*), 275
updated() (*iris.aux_factory.HybridHeightFactory method*), 277
updated() (*iris.aux_factory.HybridPressureFactory method*), 278
updated() (*iris.aux_factory.OceanSFactory method*), 280
updated() (*iris.aux_factory.OceanSg1Factory method*), 281
updated() (*iris.aux_factory.OceanSg2Factory method*), 283
updated() (*iris.aux_factory.OceanSigmaFactory method*), 284
updated() (*iris.aux_factory.OceanSigmaZFactory method*), 286
UriProtocol (*class in iris.io.format_picker*), 431
uses_weighting() (*iris.analysis.WeightedAggregator method*), 267

V

values (*iris.fileformats.name_loaders.NAMECoord attribute*), 404
values() (*iris.fileformats.cf.CFGroup method*), 396
var_name (*iris.aux_factory.AtmosphereSigmaFactory property*), 274
var_name (*iris.aux_factory.AuxCoordFactory property*), 276
var_name (*iris.aux_factory.HybridHeightFactory property*), 277
var_name (*iris.aux_factory.HybridPressureFactory property*), 279
var_name (*iris.aux_factory.OceanSFactory property*), 280
var_name (*iris.aux_factory.OceanSg1Factory property*), 282
var_name (*iris.aux_factory.OceanSg2Factory property*), 284
var_name (*iris.aux_factory.OceanSigmaFactory property*), 285
var_name (*iris.aux_factory.OceanSigmaZFactory property*), 287
var_name (*iris.common.metadata.AncillaryVariableMetadata attribute*), 290
var_name (*iris.common.metadata.BaseMetadata attribute*), 292
var_name (*iris.common.metadata.CellMeasureMetadata attribute*), 294
var_name (*iris.common.metadata.CoordMetadata attribute*), 296
var_name (*iris.common.metadata.CubeMetadata attribute*), 298
var_name (*iris.common.metadata.DimCoordMetadata attribute*), 300
var_name (*iris.common.mixin.CFVariableMixin property*), 301
var_name (*iris.coords.AncillaryVariable property*), 328
var_name (*iris.coords.AuxCoord property*), 334
var_name (*iris.coords.CellMeasure property*), 337
var_name (*iris.coords.Coord property*), 344
var_name (*iris.coords.DimCoord property*), 351
var_name (*iris.cube.Cube property*), 368
variable_name (*iris.fileformats.netcdf.NetCDFDataProxy attribute*), 408
VARIANCE (*in module iris.analysis*), 262
vector_coord() (*in module iris.fileformats.rules*), 420
vector_dims_shape (*iris.fileformats.um.FieldCollation property*), 427
verify() (*in module iris.fileformats.pp_save_rules*), 419
VerticalPerspective (*class in iris.coord_systems*), 324
vmax (*iris.palette.SymmetricNormalize property*), 437
vmin (*iris.palette.SymmetricNormalize property*), 437

W

WeightedAggregator (*class in iris.analysis*), 265
with_traceback() (*iris.exceptions.AncillaryVariableNotFoundError method*), 374

with_traceback() (*iris.exceptions.CellMeasureNotFoundError method*), 374
with_traceback() (*iris.exceptions.ConcatenateError method*), 374
with_traceback() (*iris.exceptions.ConstraintMismatchError method*), 374
with_traceback() (*iris.exceptions.CoordinateCollapseError method*), 374
with_traceback() (*iris.exceptions.CoordinateMultiDimError method*), 375
with_traceback() (*iris.exceptions.CoordinateNotFoundError method*), 375
with_traceback() (*iris.exceptions.CoordinateNotRegularError method*), 375
with_traceback() (*iris.exceptions.DuplicateDataError method*), 375
with_traceback() (*iris.exceptions.IgnoreCubeException method*), 375
with_traceback() (*iris.exceptions.InvalidCubeError method*), 376
with_traceback() (*iris.exceptions.IrisError method*), 376
with_traceback() (*iris.exceptions.LazyAggregatorError method*), 376
with_traceback() (*iris.exceptions.MergeError method*), 376
with_traceback() (*iris.exceptions.NotYetImplementedError method*), 376
with_traceback() (*iris.exceptions.TranslationError method*), 377
with_traceback() (*iris.exceptions.UnitConversionError method*), 377
with_traceback() (*iris.fileformats.netcdf.UnknownCellMethodWarning method*), 412
with_traceback() (*iris.IrisDeprecation method*), 465
WPERCENTILE (*in module iris.analysis*), 262
wrap_lons() (*in module iris.analysis.cartography*), 247
write() (*iris.fileformats.netcdf.Saver method*), 410

X

x_bounds (*iris.fileformats.pp.PPField property*), 416
xml() (*iris.cube.Cube method*), 366
xml() (*iris.cube.CubeList method*), 373
xml_element() (*iris.aux_factory.AtmosphereSigmaFactory method*), 273
xml_element() (*iris.aux_factory.AuxCoordFactory method*), 275
xml_element() (*iris.aux_factory.HybridHeightFactory method*), 277
xml_element() (*iris.aux_factory.HybridPressureFactory method*), 278
xml_element() (*iris.aux_factory.OceanSFactory method*), 280
xml_element() (*iris.aux_factory.OceanSg1Factory method*), 282
xml_element() (*iris.aux_factory.OceanSg2Factory method*), 283
xml_element() (*iris.aux_factory.OceanSigmaFactory method*), 285
xml_element() (*iris.aux_factory.OceanSigmaZFactory method*), 286
xml_element() (*iris.coord_systems.AlbersEqualArea method*), 315
xml_element() (*iris.coord_systems.CoordSystem method*), 315
xml_element() (*iris.coord_systems.GeogCS method*), 316
xml_element() (*iris.coord_systems.Geostationary method*), 317
xml_element() (*iris.coord_systems.LambertAzimuthalEqualArea method*), 318
xml_element() (*iris.coord_systems.LambertConformal method*), 319
xml_element() (*iris.coord_systems.Mercator method*), 320
xml_element() (*iris.coord_systems.Orthographic method*), 321
xml_element() (*iris.coord_systems.OSGB method*), 320
xml_element() (*iris.coord_systems.RotatedGeogCS method*), 322
xml_element() (*iris.coord_systems.Stereographic method*), 323
xml_element() (*iris.coord_systems.TransverseMercator method*), 324
xml_element() (*iris.coord_systems.VerticalPerspective method*), 325
xml_element() (*iris.coords.AncillaryVariable method*), 327
xml_element() (*iris.coords.AuxCoord method*), 333
xml_element() (*iris.coords.CellMeasure method*), 336
xml_element() (*iris.coords.CellMethod method*), 337
xml_element() (*iris.coords.Coord method*), 343
xml_element() (*iris.coords.DimCoord method*), 350

Y

y_bounds (*iris.fileformats.pp.PPField property*), 416
year (*iris.time.PartialDateTime attribute*), 448