cf-python Documentation

Release 0.9.5

David Hassell

October 01, 2012

CONTENTS

Part I

Introduction

CF[1] is a netCDF[2] convention which is in wide and growing use for the storage of model-generated and observational data relating to the atmosphere, ocean and Earth system.

This package is an implementation of the CF data model[3], and as such it is an API allows for the full scope of data and metadata interactions described by the CF conventions.

With this package you can:

- Read CF-netCDF and PP[4] format files, aggregating their contents into as few multidimensional fields as possible.
- Write fields to CF-netCDF files on disk.
- Create, delete and modify a field's data and metadata.
- Select fields by their metadata values.
- Subset a field's data to create a new field.
- Perform arithmetic and comparison operations with fields.

All of the above use *Large Amounts of Massive Arrays (LAMA)* functionality, which allows multiple fields larger than the available memory to exist and be manipulated.

See the cf-python home page[5] for downloads, installation and source code.

---

[1] http://cf-pcmdi.llnl.gov/documents/cf-conventions/1.6/cf-conventions.html
[2] http://www.unidata.ucar.edu/software/netcdf
[3] https://cf-pcmdi.llnl.gov/trac/ticket/68
[4] http://badc.nerc.ac.uk/help/formats/pp-format/
[5] http://code.google.com/p/cf-python/

Part II

Getting started

CHAPTER

ONE

A FIRST EXAMPLE

The cf package allows a data array and its associated metadata to be contained and manipulated as a single entity called a *field*, which is stored in a `Field` object.

Much of the basic field manipulation syntax can be seen in this simple read-modify-write example which:

- Reads a field from a file on disk and find out information about it.
- Modifies an attribute and the units of its data.
- Modifies the data values.
- Modifies a subset of the data values.
- Writes it out to another file on disk.

The example may be reproduced by downloading the `sample netCDF file (file.nc)` (taking care not to overwrite an existing file with that name).

1. Import the cf package.

```
>>> import cf
```

2. Read a field from disk and find a summary of its contents.

```
>>> f = cf.read('file.nc')[0]
>>> type(f)
<class 'cf.field.Field'>
>>> f
<CF Field: air_temperature(4, 5)>
>>> print f
Data            : air_temperature(latitude, longitude)
Cell methods    : time: mean
Dimensions      : time(1) = [15] days since 2000-1-1
                : latitude(4) = [-2.5, ..., 5] degrees_north
                : longitude(5) = [0, ..., 15] degrees_east
                : height(1) = [2] m
Auxiliary coords:
```

3. Find all of the field's attributes and its data array as a masked numpy array.

```
>>> f.attributes
{'Conventions': 'CF-1.0',
 '_FillValue': 1e+20,
 'cell_methods': <CF CellMethods: time: mean>,
 'experiment_id': 'stabilization experiment (SRES A1B)',
 'long_name': 'Air Temperature',
 'standard_name': 'air_temperature',
 'title': 'SRES A1B',
 'units': 'K'}
>>> f.array
masked_array(data =
```

```
 [[ 274.15,  276.15,  275.15,  277.15,  278.15],
  [ 274.15,  275.15,  276.15,  277.15,  276.15],
  [ 277.15,  275.15,  278.15,  274.15,  278.15],
  [ 276.15,  274.15,  275.15,  277.15,  274.15]],
            mask = False,
      fill_value = 1e+20)
```

4.. Modify the field's long name attribute and change the field's data from units of Kelvin to Celsius.

Note: Changing the units automatically changes the data when it is next accessed.

```
>>> f.long_name
'Air Temperature'
>>> f.long_name = 'Surface Air Temperature'
>>> f.long_name
'Surface Air Temperature'
>>> f.Units -= 273.15
>>> f.units
'K @ 273.15'
>>> f.array
masked_array(data =
 [[ 1.  3.  2.  4.  5.]
  [ 1.  2.  3.  4.  3.]
  [ 4.  2.  5.  1.  5.]
  [ 3.  1.  2.  4.  1.]],
            mask = False,
      fill_value = 1e+20)
```

5. Check that the field has 'temperature' in its standard name and that at least one of its longitude coordinate values is greater than 0.0.

```
>>> f.match(attr  = {'standard_name': '.*temperature.*'},
            coord = {'longitude'    : cf.gt(0)})
True
```

6. Modify the data values.

```
>>> g = f + 100
>>> g.array
masked_array(data =
 [[ 101.  103.  102.  104.  105.]
  [ 101.  102.  103.  104.  103.]
  [ 104.  102.  105.  101.  105.]
  [ 103.  101.  102.  104.  101.]],
            mask = False,
      fill_value = 1e+20)
>>> g = f + f
>>> g.array
masked_array(data =
 [[  2.   6.   4.   8.  10.]
  [  2.   4.   6.   8.   6.]
  [  8.   4.  10.   2.  10.]
  [  6.   2.   4.   8.   2.]],
            mask = False,
      fill_value = 1e+20)
>>> g = f / cf.Data(2, 'seconds')
>>> g.array
masked_array(data =
 [[0.5 1.5 1.0 2.0 2.5]
  [0.5 1.0 1.5 2.0 1.5]
  [2.0 1.0 2.5 0.5 2.5]
  [1.5 0.5 1.0 2.0 0.5]],
```

```
            mask = False,
       fill_value = 1e+20)
>>> g.Units
<CF Units: s-1.K>
```

7. Access and modify a subset of the data values.

```
>>> g = f.subset[0::2, 2:4]
>>> g.array
masked_array(data =
 [[ 2.  4.]
  [ 5.  1.]],
            mask = False,
       fill_value = 1e+20)
>>> f.subset[0::2, ...] = -10
>>> f.array
masked_array(data =
 [[-10. -10. -10.  -10.  -10.]
  [  1.   2.   3.    4.    3.]
  [-10. -10. -10.  -10.  -10.]
  [  3.   1.   2.    4.    1.]],
            mask = False,
       fill_value = 1e+20)
```

8. Write the modified field to disk.

```
>>> cf.write(f, 'newfile.nc')
```

CHAPTER

TWO

FURTHER EXAMPLES

2.1 Reading

The `read` function will read CF-netCDF and PP format files from disk (or netCDF file from an OPeNDAP server) and return their contents as a *field list*, i.e. an ordered collection of fields stored in a `FieldList` object:

```
>>> f = cf.read('data.nc')
>>> type(f)
<class 'cf.field.FieldList'>
>>> f
[<CF Field: pressure(30, 24)>,
 <CF Field: u_compnt_of_wind(19, 29, 24)>,
 <CF Field: v_compnt_of_wind(19, 29, 24)>,
 <CF Field: potential_temperature(19, 30, 24)>]
>>> type(f[-1])
<class 'cf.field.Field'>
>>> f[-1]
<CF Field: potential_temperature(19, 30, 24)>
```

The `read` function always returns a field list as opposed to a field. If a single field as a `Field` object is required, then the `read` function (or rather its returned field list) may be indexed:

```
>>> f = cf.read('file1.nc')[0]
>>> type(f)
<class 'cf.field.Field'>
```

See the section on *variable lists* for more details on fields and list of fields.

Multiple files may be read at once by:

- Using Unix shell wildcard characters in file names
- Providing a list or tuple of files

```
>>> f = cf.read('*.nc')
>>> f = cf.read('file[1-9a-c].nc')
>>> f = cf.read('dir*/*.pp')
>>> f = cf.read(['file1.nc', 'file2.nc'])
```

The file format is inferred from the file contents, not from the file name suffix.

2.2 Writing

The `write` function will write fields to a CF-netCDF file on disk:

```
>>> type(g)
<class 'cf.field.Field'>
>>> cf.write(g, 'newfile.nc')
```

```
>>> type(f)
<class 'cf.field.FieldList'>
>>> cf.write(f, 'newfile.nc')
```

A sequence of fields and field lists may be written to the same file:

```
>>> cf.write([f, g], 'newfile.nc')
```

All of the input fields are written to the same output file, but if metadata (such as coordinates) are identical in two or more fields then that metadata is only written once to the output file.

Output file names are arbitrary (in particular, they do not require a suffix).

2.3 Attributes

The field's standard and non-standard CF attributes are returned by the `attributes` attribute:

```
>>> f.attributes
{'_FillValue': 1e+20,
 'cell_methods': <CF CellMethods: time: mean>,
 'standard_name': 'air_temperature',
 'units': 'K'}
```

Individual attributes recognised by the CF conventions (such as standard_name) may be set, retrieved and deleted as standard python object attributes:

```
>>> f.standard_name = 'air_temperature'
>>> f.standard_name
'air_temperature'
>>> del f.standard_name
>>> setattr(f, 'standard_name', 'air_pressure')
>>> getattr(f, 'standard_name')
'air_pressure'
>>> delattr(f, 'standard_name')
```

Any attribute (recognised by the CF conventions or not) may be retrieved with the field's `getattr` method, which accepts an optional default value argument in the same way as the python built-in getattr function:

```
>>> f.getattr('standard_name')
'air_temperature'
>>> f.getattr('non_standard_attribute', None)
3.5
>>> f.hasattr('another_non_standard_attribute')
False
>>> f.getattr('another_non_standard_attribute', None)
None
```

Any attribute (recognised by the CF conventions or not) may be set with the field's `setattr` method:

```
>>> f.setattr('standard_name', 'air_temperature')
>>> f.setattr('non_standard_attribute', 3.5)
```

Any attribute (recognised by the CF conventions or not) may be deleted with the field's `delattr` method.

```
>>> f.delattr('standard_name')
>>> f.delattr('non_standard_attribute')
```

Other attributes (called *private* attributes) which do not clash with reserved names (such as 'long_name', 'match', etc.) may be set, retrieved and deleted as usual, but they will not appear in the attributes returned by the `attributes` dictionary:

```
>>> f.ncvar = 'tas'
>>> f.ncvar
>>> 'tas'
>>> del f.ncvar
>>> f.file
'file.nc'
```

2.4 Selecting fields

Fields may be selected with the `match` and `extract`. These methods take conditions on field attributes and coordinates as inputs:

```
>>> f
[<CF Field: eastward_wind(110, 106)>,
 <CF Field: air_temperature(12, 73, 96)>]
>>> f.match(attr={'standard_name': '.*temperature'})
[False, True]
>>> g = f.extract(attr={'standard_name': '.*temperature'}, coord={'longitude': 0})
>>> g
[<CF Field: air_temperature(12, 73, 96)>]
```

2.5 The data array

The fields data array may be retrieved as an independent numpy array with the field's `array` attribute:

```
>>> f.array
masked_array(data =
 [[ 2.  4.]
  [ 5.  1.]],
             mask = False,
       fill_value = 1e+20)
```

The first and last elements of the data array may be retrieved as with the field's `first_datum` and `last_datum` attributes:

```
>>> f.first_datum
2.0
>>> f.last_datum
1.0
```

2.6 Coordinates

A coordinate of the field's coordinate system is returned by the field's `coord` method. A coordinate has the same attribute and data access principles as a field:

```
>>> c = f.coord('time')
>>> c
<CF Coordinate: time(12)>
>>> c.attributes
{'_FillValue': None,
 'axis': 'T',
 'bounds': <CF CoordinateBounds: time_bnds(12, 2)>,
 'calendar': 'noleap',
 'long_name': 'time',
 'standard_name': 'time',
 'units': 'days since 0000-1-1'}
```

```
>>> c.Units
<CF Units: days since 0000-1-1 calendar=noleap>
>>> c.array
masked_array(data = [  0  30  60  90 120 150 180 210 240 270 300 330],
             mask = False,
       fill_value = 999999)
```

2.7 Creating fields

Part III

Reference manual

CHAPTER

THREE

FIELD STRUCTURE

A field (stored in a `Field` object) is a container for a data array (stored in a `Data` object) and metadata comprising properties to describe the physical nature of the data and a coordinate system (called a *space*, stored in a `Space` object), which describes the positions of each element of the data array.

It is structured in exactly the same way as a field construct defined by the CF data model[1].

The field's space may contain coordinates and cell measures (which themselves contain data arrays and properties to describe them; and are stored in `Coordinate` and `CellMeasures` objects respectively) and transforms (stored in `Transform` objects) to describe how other auxiliary coordinates may be computed.

As in the CF data model, all components of a field are optional.

Example

The structure is exposed by printing out a full dump of a field, followed by descriptions of some of the output sections:

```
>>> type(f)
<class 'cf.field.Field'>
>>> cf.dump(f)
field summary
-------------
Data             : air_temperature(time, latitude, longitude)
Cell methods     : time: mean
Dimensions       : time(12) = [15, ..., 345] days since 1860-1-1
                 : latitude(73) = [-90, ..., 90] degrees_north
                 : longitude(96) = [0, ..., 356.25] degrees_east
                 : height(1) = [2] m
Auxiliary coords:

air_temperature field
---------------------
Field.shape = (12, 73, 96)
Field.first_datum = 245.965759277
Field.last_datum  = 238.590637207
Field._FillValue = 1e+20
Field.Units = <CF Units: K>
Field.cell_methods = <CF CellMethods: time: mean>

Field.Conventions = 'CF-1.5'
Field.experiment_id = 'climate of the 20th Century experiment (20C3M)'
Field.long_name = 'Surface Air Temperature'
Field.standard_name = 'air_temperature'
Field.title = 'model output prepared for IPCC AR4'

space
-----
```

[1] http://www.met.rdg.ac.uk/ jonathan/CF_metadata/cfdm.html

```
Field.space.dimension_sizes = {'dim2': 96, 'dim3': 1, 'dim0': 12, 'dim1': 73}

Field.space.dimensions['data'] = ['dim0', 'dim1', 'dim2']
Field.space.dimensions['dim0'] = ['dim0']
Field.space.dimensions['dim1'] = ['dim1']
Field.space.dimensions['dim2'] = ['dim2']
Field.space.dimensions['dim3'] = ['dim3']

time coordinate
---------------
Field.space['dim0'].shape = (12,)
Field.space['dim0'].first_datum = 15.0
Field.space['dim0'].last_datum  = 345.0
Field.space['dim0']._FillValue = None
Field.space['dim0'].Units = <CF Units: days since 1860-1-1 calendar=360_day>

Field.space['dim0'].axis = 'T'
Field.space['dim0'].long_name = 'time'
Field.space['dim0'].standard_name = 'time'

Field.space['dim0'].bounds.shape = (12, 2)
Field.space['dim0'].bounds.first_datum = 0.0
Field.space['dim0'].bounds.last_datum  = 360.0
Field.space['dim0'].bounds._FillValue = None
Field.space['dim0'].bounds.Units = <CF Units: days since 1860-1-1 calendar=360_day>

latitude coordinate
-------------------
Field.space['dim1'].shape = (73,)
Field.space['dim1'].first_datum = -90.0
Field.space['dim1'].last_datum  = 90.0
Field.space['dim1']._FillValue = None
Field.space['dim1'].Units = <CF Units: degrees_north>

Field.space['dim1'].axis = 'Y'
Field.space['dim1'].long_name = 'latitude'
Field.space['dim1'].standard_name = 'latitude'

Field.space['dim1'].bounds.shape = (73, 2)
Field.space['dim1'].bounds.first_datum = -90.0
Field.space['dim1'].bounds.last_datum  = 90.0
Field.space['dim1'].bounds._FillValue = None
Field.space['dim1'].bounds.Units = <CF Units: degrees_north>

longitude coordinate
--------------------
Field.space['dim2'].shape = (96,)
Field.space['dim2'].first_datum = 0.0
Field.space['dim2'].last_datum  = 356.25
Field.space['dim2']._FillValue = None
Field.space['dim2'].Units = <CF Units: degrees_east>

Field.space['dim2'].axis = 'X'
Field.space['dim2'].long_name = 'longitude'
Field.space['dim2'].standard_name = 'longitude'

Field.space['dim2'].bounds.shape = (96, 2)
Field.space['dim2'].bounds.first_datum = -1.875
Field.space['dim2'].bounds.last_datum  = 358.125
Field.space['dim2'].bounds._FillValue = None
Field.space['dim2'].bounds.Units = <CF Units: degrees_east>

height coordinate
```

```
-----------------
Field.space['dim3'].shape = (1,)
Field.space['dim3'].first_datum = 2.0
Field.space['dim3']._FillValue = None
Field.space['dim3'].Units = <CF Units: m>

Field.space['dim3'].axis = 'Z'
Field.space['dim3'].long_name = 'height'
Field.space['dim3'].positive = 'up'
Field.space['dim3'].standard_name = 'height'
```

field summary

Describes the field in terms of physical quantity of its *data array* (air_temperature), the identities of its dimensions (time, latitude, longitude and height) and the ranges of coordinate values along each axis.

air temperature field

Describes the field's *data array* (array shape, first and last values, fill value, units and cell methods) and other descriptive properties (Conventions, experiment_id, long_name, standard_name and title)

space

Describes the coordinate system of the field by describing the coordinates, cell measures and transforms. See the *space_structure* section for more details.

time coordinate

Describes the coordinate's *data array* (array shape, first and last values, fill value, and units), the coordinate's cell bounds array (array shape, first and last values, fill value, and units) and other descriptive properties (axis, long_name and standard_name)

3.1 Metadata

Most metadata properties describing the data directly are stored as familiar python objects (strings, numbers, numpy arrays, etc.):

```
>>> f.standard_name
'air_temperature'
>>> f._FillValue
1e+20
>>> f.flag_values
array([0, 1, 2, 4], dtype=int8)
```

There are some metadata properties which require their own class:

Property	Class	Description
cell_methods	`CellMethods`	The characteristics that are is represented by cell values
Flags	`Flags`	The self describing CF flag values, meanings and masks
Units	`Units`	The units of the data array

```
>>> f.cell_methods
<CF CellMethods: time: mean (interval: 1.0 month)>
>>> f.Flags
<CF Flags: values=[0 1 2], masks=[0 2 2], meanings=['low' 'medium' 'high']>
>>> f.Units
<CF Units: days since 1860-1-1 calendar=360_day>
```

The `Units` object may be accessed through the field's `units` and `calendar` attributes and the `Flags` object may be accessed through the field's `flag_values`, `flag_meanings` and `flag_masks` attributes:

```
>>> f.calendar = 'noleap'
>>> f.flag_values = ['a', 'b', 'c']
```

The Units and Flags objects may also be manipulated directly, which automatically adjusts the relevant CF attributes:

```
>>> f.Units
<CF Units: 'm'>
>>> f.units
'm'
>>> f.Units *= 1000
>>> f.Units
<CF Units: '1000 m'>
>>> f.units
'1000 m'
>>> f.Units.units = '10 m'
>>> f.units
'10 m'
```

3.2 Space structure

A space completely describes the field's coordinate system.

It contains the dimension constructs, auxiliary coordinate constructs, transform constructs and cell measure constructs defined by the CF data model[2].

A field's space is stored in its space attribute, the value of which is a Space object.

The space is a dictionary-like object whose key/value pairs identify and store the coordinate and cell measure constructs which describe it.

3.2.1 Dimensionality

The dimension sizes of the space are given by the space's dimension_sizes attribute.

```
>>> f.space.dimension_sizes
{'dim1': 19, 'dim0': 12, 'dim2': 73, 'dim3': 96}
```

3.2.2 Components

The space's key/value pairs identify and store its coordinate (Coordinate) and cell measure (CellMeasures) constructs.

Keys for dimension, auxiliary coordinate and cell measure constructs are prefixed "dim", "aux" and "cm" respectively and followed by arbitrary, unique integers for discrimination:

```
>>> f.space['dim0']
<CF Coordinate: time(12)>
>>> f.space['dim2']
<CF Coordinate: latitude73)>
>>> f.space['aux0']
<CF Coordinate: forecast_time(12)>
```

The dimensions of each of these components, and of the field's data array, are stored as ordered lists in the dimensions attribute:

```
>>> f.space.dimensions
{'data': ['dim0', 'dim1', 'dim2', 'dim3'],
 'aux0': ['dim0'],
 'dim0': ['dim0'],
 'dim1': ['dim1'],
```

[2] http://www.met.rdg.ac.uk/ jonathan/CF_metadata/cfdm.html

```
 'dim2': ['dim2'],
 'dim3': ['dim3']}
```

Note: The field's data array may contain fewer size 1 dimensions than its space.

Transform constructs are stored in the `transforms` attribute, which is a dictionary-like object containing `Transform` objects.

```
>>> f.space.transforms
{'trans0': <CF Transform: atmosphere_sigma_coordinate>}
```

Note: A single transform construct may be associated with any number of the space's coordinates via their `transform` attributes

---

3.3 Field list

A `FieldList` object is an ordered sequence of fields analogous to a built-in python list.

It has all of the *python list-like methods*[3] (`__contains__`, `__getitem__`, `__setitem__`, `__len__`, `__delitem__`, `append`, `count`, `extend`, `index`, `insert`, `pop`, `remove`, `reverse`), which behave as expected. For example:

```
>>> type(fl)
<class 'cf.field.FieldList'>
>>> fl
[<CF Field: eastward_wind(110, 106)>,
 <CF Field: air_temperature(12, 73, 96)>]
>>> len(fl)
2
>>> for f in fl:
...     print repr(f)
...
<CF Field: eastward_wind(110, 106)>,
<CF Field: air_temperature(12, 73, 96)>
>>> for f in fl[::-1]:
...     print repr(f)
...
<CF Field: air_temperature(12, 73, 96)>
<CF Field: eastward_wind(110, 106)>,
>>> f = fl[0]
>>> type(f)
<class 'cf.field.Field'>
>>> f in fl
True
>>> f = fl.pop()
>>> type(f)
<class 'cf.field.Field'>
```

3.4 Field versus field list

In some contexts, whether an object is a field or a field list is not known and does not matter. So to avoid ungainly type testing, some aspects of the `FieldList` interface are shared by a `Field` and vice versa.

[3] http://docs.python.org/2.7/reference/datamodel.html#sequence-types

3.4.1 Attributes and methods

Any attribute or method belonging to a field may be used on a field list and will be applied independently to each element:

```
>>> fl.ndim
[2, 3]
>>> fl.subset[..., 0]
[<CF Field: eastward_wind(110, 1)>,
 <CF Field: air_temperature(12, 73, 1)>]
>>> fl **= 2
[<CF Field: eastward_wind**2(110, 106)>,
 <CF Field: air_temperature**2(12, 73, 1)>]
>>> fl.squeeze('longitude')
[<CF Field: eastward_wind**2(110, 1)>,
 <CF Field: air_temperature**2(12, 73)>]
```

Standard and non-standard CF attributes may be changed to a common value with the `setattr` method:

```
>>> fl.setattr('comment', 'my data')
>>> fl.comment
['my data', 'my data']
>>> fl.setattr('foo', 'bar')
>>> fl.getattr('foo')
['bar', 'bar']
```

Changes tailored to each individual field in the list need to be carried out in a loop:

```
>>> for f in fl:
...     f.long_name = f.long_name.upper()

>>> long_names = ('square of eastward wind', 'square of temperature')
>>> for f, value in zip(fl, long_names):
...     f.long_name = value
```

3.4.2 Looping

Just as it is straight forward to iterate over the fields in a field list, a field will behave like a single element field list in iterative and indexing contexts:

```
>>> f
<CF Field: air_temperature(12, 73, 96)>
>>> for g in f:
...     repr(g)
...
<CF Field: air_temperature(12, 73, 96)>
```

CHAPTER

FOUR

UNITS

A field (as well as any other object which *inherits* from `Variable`) always contains a `Units` object which gives the physical units of the values contained in its data array.

The `Units` object is stored in the field's *Units* attribute but may also be accessed through the field's `units` and `calendar` attributes, which may take any value allowed by the CF conventions[1]. In particular, the value of the `units` attribute is a string that can be recognized by UNIDATA's Udunits-2 package[2], with a few exceptions for greater consistency with CF. These are detailed by `Units` object.

4.1 Assignment

The Field's units may be assigned directly to its `Units` object:

```
>>> f.Units.units = 'days since 1-1-1'
>>> f.Units.calendar = 'noleap'

>>> f.Units = cf.Units('metre')
```

But the same result is achieved by assigning to the field's `units` and `calendar` attributes:

```
>>> f.units = 'days since 1-1-1'
>>> f.calendar = 'noleap'
>>> f.Units
<CF Units: days since 1-1-1 calendar=noleap>
>>> f.units
'days since 1-1-1'
>>> f.calendar
'noleap'
```

4.2 Time units

Time units may be given as durations of time or as an amount of time since a reference time:

```
>>> f.units = 'day'
>>> f.units = 'seconds since 1992-10-8 15:15:42.5 -6:00'
```

Note: It is recommended that the units `year` and `month` be used with caution, as explained in the following excerpt from the CF conventions: "The Udunits package defines a year to be exactly 365.242198781 days (the interval between 2 successive passages of the sun through vernal equinox). It is not a calendar year. Udunits includes the following definitions for years: a common_year is 365 days, a leap_year is 366 days, a Julian_year is 365.25 days, and a Gregorian_year is 365.2425 days. For similar reasons the unit `month`, which is defined to be exactly year/12, should also be used with caution."

[1] http://cf-pcmdi.llnl.gov/documents/cf-conventions/latest-cf-conventions-document-1
[2] http://www.unidata.ucar.edu/software/udunits/

4.2.1 Calendar

The date given in reference time units is always associated with one of the calendars recognized by the CF conventions and may be set with the *calendar* attribute (on the field or units object).

If the calendar is not set then, as in the CF conventions, for the purposes of calculation and comparison, it defaults to the mixed Gregorian/Julian calendar as defined by Udunits:

```
>>> f.units = 'days since 2000-1-1'
>>> f.calendar
AttributeError: Can't get 'Field' attribute 'calendar'
>>> g.units = 'days since 2000-1-1'
>>> g.calendar = 'gregorian'
>>> g.Units.equals(f.Units)
True
```

The calendar is ignored for units other than reference time units.

4.3 Changing units

Changing units to equivalent units causes the variable's data array values to be modified in place (if required) when they are next accessed, and not before:

```
>>> f.units
'metre'
>>> f.array
array([    0.,  1000.,  2000.,  3000.,  4000.])
>>> f.units = 'kilometre'
>>> f.units
'kilometre'
>>> f.array
array([ 0.,  1.,  2.,  3.,  4.])

>>> f.units
'hours since 2000-1-1'
>>> f.array
array([-1227192., -1227168., -1227144.])
>>> f.units = 'days since 1860-1-1'
>>> f.array
array([ 1.,  2.,  3.])
```

The `Units` object may be operated on with augmented arithmetic assignments and binary arithmetic operations:

```
>>> f.units
'kelvin'
>>> f.array
array([ 273.15,  274.15,  275.15,  276.15,  277.15])

>>> f.Units -= 273.15
>>> f.units
'K @ 273.15'
>>> f.array
array([ 0.,  1.,  2.,  3.,  4.])

>>> f.Units = f.Units + 273.15
>>> f.units
'K'
>>> f.array
array([ 273.15,  274.15,  275.15,  276.15,  277.15])
```

```
>>> f.units = 'K @ 237.15'
'K @ 273.15'
>>> f.array
array([ 0.,  1.,  2.,  3.,  4.])
```

If the field has a data array and its units are changed to non-equivalent units then a `TypeError` will be raised when the data are next accessed:

```
>>> f.units
'm s-1'
>>> f.units = 'K'
>>> f.array
TypeError: Units are not convertible: <CF Units: m s-1>, <CF Units: K>
```

4.3.1 Changing the calendar

Warning: Do not change the calendar of reference time units in the current version. Whilst this is possible, it will almost certainly result in an incorrect interpretation of the data array or an error. Allowing the calendar to be changed is under development and will be available soon.

4.4 Equality and equivalence of units

The `Units` object has methods for assessing whether two units are equivalent or equal, regardless of their exact string representations.

Two units are equivalent if and only if numeric values in one unit are convertible to numeric values in the other unit (such as `kilometres` and `metres`). Two units are equal if and only if they are equivalent and their conversion is a scale factor of 1 (such as `kilometres` and `1000 metres`). Note that equivalence and equality are based on internally stored binary representations of the units, rather than their string representations.

```
>>> f.units = 'm/s'

>>> g.units = 'm s-1'
>>> f.Units == g.Units
True
>>> f.Units.equals(g.Units)
True

>>> g.units = 'km s-1'
>>> f.Units.equivalent(g.Units)
False

>>> f.units = 'days since 1987-12-3'
>>> g.units = 'hours since 2000-12-1'
>>> f.Units == g.Units
False
>>> f.Units.equivalent(g.Units)
True
```

4.5 Coordinate units

The units of a coordinate's bounds are always the same as the coordinate itself, and the units of the bounds automatically change when a coordinate's units are changed:

```
>>> c.units
'degrees'
>>> c.bounds.units
'degrees'
>>> c.bounds.array
array([  0.,  90.])
>>> c.units = 'radians'
>>> c.bounds.units
'radians'
>>> c.bounds.array
array([ 0.        ,  1.57079633])
```

CHAPTER

FIVE

FIELD MANIPULATION

Manipulating a field generally involves operating on its data array and making any necessary changes to the field's space to make it consistent with the new array.

5.1 Data array

A field's data array is stored by the *Data* attribute as a `Data` object:

```
>>> type(f.Data)
<class 'cf.data.Data'>
```

This `Data` object:

- Contains an N-dimensional array with many similarities to a *numpy array*[1]
- Contains the *units* of the array elements.
- Uses *LAMA functionality* to store and operate on arrays which are larger then the available memory.
- Supports masked arrays [2], regardless of whether or not it was initialized with a masked array.

5.1.1 Data mask

The data array's mask may be retrieved and deleted with the field's `mask` attribute. The mask is returned as a `Data` object:

```
>>> f.shape
(12, 73, 96)
>>> m = f.mask
>>> type(m)
<cf.data.Data>
>>> m.dtype
dtype('bool')
>>> m.shape
[12, 73, 96]
>>> m.array.shape
(12, 73, 96)

>>> del f.mask
>>> f.array.mask
False
>>> import numpy
>>> f.array.mask is numpy.ma.nomask
True
```

[1] http://docs.scipy.org/doc/numpy/reference/arrays.html#arrays

[2] Arrays that may have missing or invalid entries

5.1.2 Conversion to a numpy array

A field's data array may be converted to either a numpy array view (`numpy.ndarray.view`[3]) or an independent numpy array of the underlying data with the `varray` and `array` attributes respectively:

```
>>> a = d.array
>>> type(a)
<class 'numpy.ndarray'>
>>> v = d.varray
>>> type(v)
<class 'numpy.ndarray'>
>>> type(v.base)
<class 'numpy.ndarray'>
```

Changing a numpy array view in place will also change the data array:

```
>>> d.array
array([1, 2, 3])
>>> v = d.varray
>>> v[0] = -999
>>> d.array
array([-999,    2,    3])
```

Warning: The numpy array created with the *array* or *varray* attribute forces all of the data to be read into memory at the same time, which may not be possible for very large arrays.

5.2 Copying

A deep copy of a variable may be created with its `copy` method or equivalently with the `copy.deepcopy`[4] function:

```
>>> g = f.copy()
>>> import copy
>>> g = copy.deepcopy(f)
```

Copying utilizes *LAMA copying functionality*.

5.3 Subsetting

Subetting a field means subsetting its data array and its space in a consistent manner.

A field may be subsetted with its `subset` attribute. This attribute may be **indexed** to select a subset from dimension index values (`f.subset[indices]`) or **called** to select a subset from dimension coordinate array values (`f.subset(coordinate_values)`):

```
>>> g = f.subset[0, ...]
>>> g = f.subset(latitude=30, longitude=cf.inside(0, 90, 'degrees'))
```

The result of subsetting a field is a new field whose data array and, crucially, any data arrays within the field's metadata (such as coordinates, for example) are subsets of their originals:

```
>>> print f
Data            : air_temperature(time, latitude, longitude)
Cell methods    : time: mean
Dimensions      : time(12) = [15, ..., 345] days since 1860-1-1
                : latitude(73) = [-90, ..., 90] degrees_north
```

[3] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.view.html#numpy.ndarray.view
[4] http://docs.python.org/2.7/library/copy.html#copy.deepcopy

```
                : longitude(96) = [0, ..., 356.25] degrees_east
                : height(1) = [2] m
Auxiliary coords:
>>> g = f.subset[-1, :, 48::-1]
>>> print g
Data            : air_temperature(time, latitude, longitude)
Cell methods    : time: mean
Dimensions      : time(1) = [345] days since 1860-1-1
                : latitude(73) = [-90, ..., 90] degrees_north
                : longitude(49) = [180, ..., 0] degrees_east
                : height(1) = [2] m
Auxiliary coords:
```

The new subsetted field is independent of the original. Subsetting utilizes *LAMA subsetting functionality*.

5.3.1 Indexing

Subsetting by dimension indices uses an extended Python slicing syntax, which is similar *numpy array indexing*[5]. There are two extensions to the numpy indexing functionality:

- **Size 1 dimensions are never removed.**

  An integer index *i* takes the *i*-th element but does not reduce the rank of the output array by one.

- **When advanced indexing is used on more than one dimension, the advanced indices work independently.**

  When more than one dimension's slice is a 1-d boolean array or 1-d sequence of integers, then these indices work independently along each dimension (similar to the way vector subscripts work in Fortran), rather than by their elements.

```
>>> print f
Data            : air_temperature(time, latitude, longitude)
Cell methods    : time: mean
Dimensions      : time(12) = [15, ..., 345] days since 1860-1-1
                : latitude(73) = [-90, ..., 90] degrees_north
                : longitude(96) = [0, ..., 356.25] degrees_east
                : height(1) = [2] m
Auxiliary coords:
>>> f.shape
(12, 73, 96)
>>> f.subset[...].shape
(12, 73, 96)
>>> f.subset[0].shape
(1, 73, 96)
>>> f.subset[0,...].shape
(1, 73, 96)
>>> f.subset[::-1].shape
(12, 73, 96)
>>> f.subset[0:5, ..., slice(None, None, 2)].shape
(5, 73, 48)
>>> lon = f.coord('longitude').array
>>> f.subset[..., lon<90]
(12, 73, 24)
>>> f.subset[[1,2], [1,2,3], [1,2,3,4]].shape
(2, 3, 4)
```

Note that the indices of the last example above would raise an error when given to a numpy array.

[5] http://docs.scipy.org/doc/numpy/reference/arrays.indexing.html#arrays-indexing

5.3.2 Coordinate values

Subsetting by coordinate values allows a subsetted field to be defined by particular coordinate values of its space.

Subsetting by coordinate values is functionally equivalent to subsetting by *indexing* – internally, the selected coordinate values are in fact converted to dimension indices.

Coordinate values are provided as arguments to a **call** to the *subset* method.

The benefits to subsetting in this fashion are:

- The dimensions to be subsetted are identified by name.
- The position in the data array of each dimension need not be known.
- Dimensions for which no subsetting is required need not be specified.
- Size 1 dimensions of the space which are not spanned by the data array may be specified.

```
>>> print f
Data            : air_temperature(time, latitude, longitude)
Cell methods    : time: mean
Dimensions      : time(12) = [15, ..., 345] days since 1860-1-1
                : latitude(73) = [-90, ..., 90] degrees_north
                : longitude(96) = [0, ..., 356.25] degrees_east
                : height(1) = [2] m
Auxiliary coords:
>>> f.subset(latitude=0).shape
(12, 1, 96)
>>> f.subset(latitude=cf.inside(-30, 30)).shape
(12, 25, 96)
>>> f.subset(longitude=cf.ge(270, 'degrees_east'), latitude=[0, 2.5, 10]).shape
(12, 3, 24)
>>> f.subset(latitude=cf.lt(0, 'degrees_north'))
(12, 36, 96)
>>> f.subset(latitude=[cf.lt(0, 'degrees_north'), 90])
(12, 37, 96)
>>> import math
>>> f.subset(longitude=cf.lt(math.pi, 'radian'), height=2)
(12, 73, 48)
>>> f.subset(height=cf.gt(3))
IndexError: No indices found for 'height' values gt 3
```

Note that if a comparison function (such as `inside`) does not specify any units, then the units of the named coordinate are assumed.

5.4 Selection

Fields may be tested for matching given conditions and selected according to those matches with the `match` and `extract` methods. Conditions may be given on:

- The field's standard and non-standard attributes (*attr* parameter).
- Any other of the field's attributes (*priv* parameter).
- The field's coordinate values (*coord* parameter).
- The field's coordinate cell sizes (*cellsize* parameter).

```
>>> f
<CF Field: air_temperature(12, 73, 96)>
>>> f.match(priv={'ncvar': 'tas'})
True
>>> g = f.extract(priv={'ncvar': 'tas'})
```

```
>>> g is f
True

>>> f
[<CF Field: eastward_wind(110, 106)>,
 <CF Field: air_temperature(12, 73, 96)>]
>>> f.match(attr={'standard_name': '.*temperature'})
[False, True]
>>> g = f.extract(attr={'standard_name': '.*temperature'}, coord={'longitude': 0})
>>> g
[<CF Field: air_temperature(12, 73, 96)>]
```

All of these keywords may be used with the `read` function to select on input:

```
>>> f = cf.read('file*.nc', attr={'standard_name': '.*temperature'}, coord={'longitude': 0})
```

5.5 Aggregation

Fields are aggregated into as few multidimensional fields as possible with the `aggregate` function:

```
>>> f
[<CF Field: eastward_wind(110, 106)>,
 <CF Field: eastward_wind(110, 106)>,
 <CF Field: eastward_wind(110, 106)>,
 <CF Field: eastward_wind(110, 106)>,
 <CF Field: eastward_wind(110, 106)>,
 <CF Field: eastward_wind(110, 106)>,
 <CF Field: air_temperature(12, 73, 96)>,
 <CF Field: air_temperature(96, 73)>]
>>> cf.aggregate(f)
>>> f
[<CF Field: eastward_wind(3, 2, 110, 106)>,
 <CF Field: air_temperature(13, 73, 96)>]
```

By default, the the fields return from the `read` function have been aggregated:

```
>>> f = cf.read('file*.nc')
>>> len(f)
1
>>> f = cf.read('file*.nc', aggregate=False)
>>> len(f)
12
```

Aggregation implements the CF aggregation rules[6].

5.6 Assignment

In-place assignment to a field's data array may be done by assigning to the field's indexed `subset` attribute, observing the `numpy broadcasting rules`[7].

Assigning to a subset uses *LAMA functionality*, so it is possible to assign to subsets which are larger than the available memory.

```
>>> print f
Data            : air_temperature(time, latitude, longitude)
Cell methods    : time: mean
Dimensions      : time(12) = [15, ..., 345] days since 1860-1-1
```

[6] http://www.met.reading.ac.uk/ david/cf_aggregation_rules.html
[7] http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html#numpy.doc.broadcasting

```
                : latitude(73) = [-90, ..., 90] degrees_north
                : longitude(96) = [0, ..., 356.25] degrees_east
                : height(1) = [2] m
Auxiliary coords:

>>> f.subset[0] = 273.15
>>> f.subset[0, 0] = 273.15
>>> f.subset[0, 0, 0] = 273.15
>>> f.subset[11, :, :] = numpy.arange(96)
```

In-place assignment may also be done by creating and then changing a numpy array view (`numpy.ndarray.view`[8]) of the data with the `varray` attribute:

```
>>> f.subset[0, 0, 0].array.item()
287.567
>>> a = f.varray
>>> type(a)
<type 'numpy.ndarray'>
>>> a[0, 0, 0] = 300
>>> f.first_datum
300.0
```

Warning: The numpy array created with the *varray* attribute forces all of the data to be read into memory at the same time, which may not be possible for very large arrays.

5.7 Arithmetic and comparison

Arithmetic and comparison operations on a field are defined as element-wise operations on the field's data array, and return a field as the result:

- When using a field in unary or binary arithmetic operations (such as `abs()`, `+` or `**`) a new, independent field is created with a modified data array.
- When using a field in augmented arithmetic operations (such as `-=`), the field's data array is modified in place.
- When using a field in comparison operations (such as `<` or `!=`) a new, independent field is created with a boolean data array.

A field's data array is modified in a very similar way to how a numpy array would be modified in the same operation, i.e. *broadcasting* ensures that the operands are compatible and the data array is modified element-wise.

Broadcasting is metadata-aware and will automatically account for arbitrary configurations, such as dimension order, but will not allow incompatible metadata to be combined, such as adding a field of height to one of temperature.

The *resulting field's metadata* will be very similar to that of the operands which are also fields. Differences arise when the existing metadata can not correctly describe the newly created field. For example, when dividing a field with units of *metres* by one with units of *seconds*, the resulting field will have units of *metres/second*.

Arithmetic and comparison utilizes *LAMA functionality* so data arrays larger than the available physical memory may be operated on.

5.7.1 Broadcasting

The term broadcasting describes how data arrays of the operands with different shapes are treated during arithmetic and comparison operations. Subject to certain constraints, the smaller array is "broadcast" across the larger array so that they have compatible shapes.

[8] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.view.html#numpy.ndarray.view

The general broadcasting rules are similar to the broadcasting rules implemented in numpy[9], the only difference being when both operands are fields, in which case the fields are temporarily conformed so that:

- Dimensions are aligned according to the coordinates' metadata to ensure that matching dimensions are broadcast against each other.
- Common dimensions have matching units.
- Common dimensions have matching axis directions.

This restructuring of the field ensures that the matching dimensions are broadcast against each other.

Broadcasting is done without making needless copies of data and so is usually very efficient.

5.7.2 What a field may be combined with

A field may be combined or compared with the following objects:

Object	Description
int[10], long[11], float[12]	The field's data array is combined with the python scalar
Data with size 1	The field's data array is combined with the Data object's scalar value, taking into account: • Different but equivalent units
Field	The two field's must satisfy the field combination rules. The fields' data arrays and spaces are combined taking into account: • Identities of dimensions • Different but equivalent units • Different dimension orders • Different dimension directions

A field may appear on the left or right hand side of an operator, but note the following warning:

Warning: Combining a numpy array on the *left* with a field on the right does work, but will give generally unintended results – namely a numpy array of fields.

5.7.3 Resulting metadata

When creating any new field, the field's *history* attribute is updated to record the operation. The fields existing name-like attributes may also need to be changed:

```
>>> f.standard_name
'air_temperature'
>>> f += 2
>>> f.standard_name
'air_temperature'
>>> f.history
'air_temperature+2'

>>> f.standard_name
'air_temperature'
>>> f **= 2
>>> f.standard_name
AttributeError: 'Field' object has no attribute 'standard_name'
>>> f.history
'air_temperature**2'
```

[9] http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html#numpy.doc.broadcasting

```
>>> f.standard_name, g.standard_name
('air_temperature', 'eastward_wind')
>>> h = f * g
>>> h.standard_name
AttributeError: 'Field' object has no attribute 'standard_name'
>>> h.long_name
'air_temperature*eastward_wind'
>>> h.history
'air_temperature*eastward_wind'
```

When creating a new field which has different physical properties to the input field(s) the units will also need to be changed:

```
>>> f.units
'K'
>>> f += 2
>>> f.units
'K'

>>> f.units
'K'
>>> f **= 2
>>> f.units
'K2'

>>> f.units, g.units
('m', 's')
>>> h = f / g
>>> h.units
'm/s'
```

When creating a new field which has a different space to the input fields, the new space will in general contain the superset of dimensions from the two input fields, but may not have some of either input field's auxiliary coordinates or size 1 dimension coordinates. Refer to the field combination rules for details.

5.7.4 Overloaded operators

A field defines the following overloaded operators for arithmetic and comparison.

Rich comparison operators

Operator	Method
<	__lt__()
<=	__le__()
==	__eq__()
!=	__ne__()
>	__gt__()
>=	__ge__()

Binary arithmetic operators

Operator	Methods	
+	__add__()	__radd__()
-	__sub__()	__rsub__()
*	__mul__()	__rmul__()
/	__div__()	__rdiv__()
/	__truediv__()	__rtruediv__()
//	__floordiv__()	__rfloordiv__()
%	__mod__()	__rmod__()
divmod()	__divmod__()	__rdivmod__()
**, pow()	__pow__()	__rpow__()
&	__and__()	__rand__()
^	__xor__()	__rxor__()
|	__or__()	__ror__()

Augmented arithmetic operators

Operator	Method
+=	__iadd__()
-=	__isub__()
*=	__imul__()
/=	__idiv__()
/=	__itruediv__()
//=	__ifloordiv__()
%=	__imod__()
**=	__ipow__()
&=	__iand__()
^=	__ixor__()
|=	__ior__()

Unary arithmetic operators

Operator	Method
-	__neg__()
+	__pos__()
abs()	__abs__()
~	__invert__()

5.8 Manipulation routines

A field has attributes and methods which return information about its data array or manipulate the data array in some manner. Many of these behave similarly to their numpy counterparts with the same name but, where appropriate, return `Field` objects rather than numpy arrays.

Attributes

Field attribute	Description	Numpy counterpart
`size`	Number of elements in the data array	`numpy.ndarray.size`[13]
`shape`	Tuple of the data array's dimension sizes	`numpy.ndarray.shape`[14]
`ndim`	Number of dimensions in the data array	`numpy.ndarray.ndim`[15]
`dtype`	Numpy data-type of the data array	`numpy.ndarray.dtype`[16]

[13] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.size.html#numpy.ndarray.size
[14] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.shape.html#numpy.ndarray.shape
[15] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.ndim.html#numpy.ndarray.ndim
[16] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.dtype.html#numpy.ndarray.dtype

Methods

Field method	Description	Numpy counterpart
expand_dims	Expand the shape of the data array	numpy.expand_dims[17]
reverse_dims	Reverse the directions of data array axes	
squeeze	Remove size 1 dimensions from the field's data array	numpy.squeeze[18]
transpose	Permute the dimensions of the data array	numpy.transpose[19]
unsqueeze	Insert size 1 dimensions from the field's space into its data array	

5.9 Manipulating other variables

Subsetting, assignment, arithmetic and comparison operations on other Variable types (such as Coordinate, CoordinateBounds, CellMeasures) are very similar to those for fields.

In general, different dimension identities, different dimension orders and different dimension directions are not considered, since these objects do not contain the coordinate system required to define these properties (unlike a field).

5.9.1 Coordinates

Coordinates are a special case as they may contain a data array for their coordinate bounds which needs to be treated consistently with the main coordinate array:

```
>>> type(c)
<cf.coordinate.Coordinate>
>>> type(c.bounds)
<cf.coordinate.CoordinateBounds>
>>> c.shape
(12,)
>>> c.bounds.shape
(12, 2)
>>> d = c.subset[0:6]
>>> d.shape
(6,)
>>> d.bounds.shape
(6, 2)
```

Warning: If the coordinate bounds are operated on directly, consistency with the parent coordinate may be broken.

[17] http://docs.scipy.org/doc/numpy/reference/generated/numpy.expand_dims.html#numpy.expand_dims
[18] http://docs.scipy.org/doc/numpy/reference/generated/numpy.squeeze.html#numpy.squeeze
[19] http://docs.scipy.org/doc/numpy/reference/generated/numpy.transpose.html#numpy.transpose

CHAPTER

SIX

LARGE AMOUNTS OF MASSIVE ARRAYS (LAMA)

Data arrays contained within fields are stored and manipulated in a very memory efficient manner such that large numbers of fields may co-exist and be manipulated regardless of the size of their data arrays. The limiting factor on array size is not the amount of available physical memory, but the amount of disk space available, which is generally far greater.

The basic functionality is:

- **Arrays larger than the available physical memory may be created.**

  Arrays larger than a preset number of bytes are partitioned into smaller sub-arrays which are not kept in memory but stored on disk, either as part of the original file they were read in from or as newly created temporary files, whichever is appropriate. Therefore data arrays larger than a preset size need never wholly exist in memory.

- **Large numbers of arrays which are in total larger than the available physical memory may co-exist.**

  Large numbers of data arrays which are collectively larger than the available physical memory may co-exist, regardless of whether any individual array is in memory or stored on disk.

- **Arrays larger than the available physical memory may be operated on.**

  Array operations (such as subsetting, assignment, arithmetic, comparison, collapses, etc.) are carried out on a partition-by-partition basis. When an operation traverses the partitions, if a partition is stored on disk then it is returned to disk before processing the next partition.

- **The memory management does not need to be known at the API level.**

  As the array's metadata (such as size, shape, data-type, number of dimensions, etc.) always reflect the data array in its entirety, regardless of how the array is partitioned and whether or not its partitions are in memory, fields and other variables containing data arrays may be used in the API as if they were normal, in-memory objects (like numpy arrays). Partitioning does carry a speed overhead, but this may be minimised for particular applications or hardware configurations.

6.1 Reading from files

When a field is read from a file, the data array is not realized in memory, however large or small it may be. Instead each partition refers to part of the original file on disk. Therefore reading even very large fields is initially very fast and uses up only a very small amount of memory.

6.2 Copying

When a field is deep copied with its `copy` method or the `copy.deepcopy`[1] function, the partitions of its data array are transferred to the new field as object identities and are *not* deep copied. Therefore copying even very large fields is initially very fast and uses up only a very small amount of memory.

The independence of the copied field is, however, preserved since each partition that is stored in memory (as opposed to on disk) is deep copied if and when the data in the new field is actually accessed, and then only if the partition's data still exists in the original (or any other) field.

6.3 Aggregation

When two fields are aggregated to form one, larger field there is no need for either field's data array partitions to be accessed, and therefore brought into memory if they were stored on disk. The resulting field recombines the two fields' array partitions as object identities into the new larger array. Therefore creating an aggregated field that uses up only a very small amount of extra memory.

The independence of the new field is, however, preserved since each partition that is stored in memory (as opposed to on disk) is deep copied when the data in the new field is actually accessed, and then only if the partition's data still exists in its the original (or any other) field.

6.4 Subsetting

When a new field is created by *subsetting* a field, the new field is actually a *LAMA deep copy* of the original but with additional instructions on each of its data partitions to only use the part specified by the subset indices.

As with copying, creating subsetted fields is initially very fast and uses up only a very small amount of memory, with the added advantage that a deep copy of only the requested parts of the data array needs to be carried out at the time of data access, and then only if the partition's data still exists in the original field.

When subsetting a field that has previously been subsetted but has not yet had its data accessed, the new subset merely updates the instructions on which parts of the array partitions to use. For example:

```
>>> f.shape = (12, 73, 96)
>>> g = f.subset[::-2, ...]
>>> h = g.subset[2:5, ...]
```

is equivalent to

```
>>> h = f.subset[7:2:-2, ...]
```

and if all of the partitions of field `f` are stored on disk then in both cases so are all of the partitions of field `h` and no data has been read from disk.

6.5 Speed and memory management

The creation of temporary files for array partitions of large arrays and the reading of data from files on disk can create significant speed overheads (for example, recent tests show that writing a 100 megabyte array to disk can take O(1) seconds), so it may be desirable to configure the maximum size of array which is kept in memory, and therefore has fast access.

The data array memory management is configurable with the *chunk size* and *free memory threshold* parameters.

[1] http://docs.python.org/2.7/library/copy.html#copy.deepcopy

- The **chunk size parameter** sets the size in bytes (default 104857600) for which field data arrays larger than that size are partitioned into sub-arrays (each of which is smaller than the chunk size) and either retained in their original input file or stored in temporary files on disk, whichever is appropriate. It is found and set with the CHUNKSIZE function.
- The **free memory threshold parameter** is the minimum amount of memory which should always be kept for temporary work space. If the amount of available memory goes below this threshold then all new field data arrays, even those smaller than the chunk size, will be stored on disk. If the free memory subsequently increases sufficiently then new field data arrays smaller than the chunk size will be allowed to remain in memory with no partitioning.

  The parameter's value is in kibibytes (1 kibibyte = 1024 bytes), but it is set by specifying an integer amount (default 10) of chunk sizes to be kept free. This asymmetry is useful since setting this parameter should be in the context of how many array partitions may need to be temporarily realized in memory at any one time. It is found and set with the FM_THRESHOLD function, but setting the chunk size with CHUNKSIZE will also update free memory threshold.

```
>>> cf.CHUNKSIZE()
104857600
>>> cf.FM_THRESHOLD()
1024000.0                 # = 104857600 * 10 / 1024.
>>> cf.CHUNKSIZE(2**30)
>>> cf.CHUNKSIZE()
1073741824
>>> cf.FM_THRESHOLD()
10485760.0                # = 1073741824 * 10 / 1024.
>>> cf.FM_THRESHOLD(20)
>>> cf.FM_THRESHOLD()
20971520.0                # = 1073741824 * 20 / 1024.
```

The default number of chunk sizes to be kept free is set with the *MINNCFM* parameter in the CONSTANTS dictionary:

```
>>> cf.CONSTANTS['MINNCFM'] = 15
```

Setting MINNCFM in this way does not automatically update the free memory threshold, but the new value will be used to calculate a new free memory threshold when the chunk size is set in with subsequent CHUNKSIZE calls.

6.5.1 Temporary files

The directory in which temporary files is found and set with the *TEMPDIR* parameter in the CONSTANTS dictionary

```
>>> cf.CONSTANTS['TEMPDIR']
'/tmp'
>>> cf.CONSTANTS['TEMPDIR'] = '/home/me/tmp'
```

The removal of temporary files which are no longer required works in the same way as python's automatic garbage collector.

When a partition's data is stored in a temporary file, that file will only exist for as long as there are partitions referring to it. When no partitions require the file it will be deleted automatically.

When python exits normally, all temporary files are always deleted.

Note that changing the temporary file directory does prevent temporary files in the original directory from being garbage collected.

6.5.2 Partitioning

To maximise looping efficiency, array partitioning preserves as much as is possible the faster varying (inner) dimensions' sizes in each of the sub-arrays .

Examples

If an array with shape (2, 4, 6) is partitioned into 2 partitions then both sub-arrays will have shape (1, 4, 6).

If the same array is partitioned into 4 partitions then all four sub-arrays will have shape (1, 2, 6).

If the same array is partitioned into 8 partitions then all eight sub-arrays will have shape (1, 2, 3).

If the same array is partitioned into 48 partitions then all forty eight sub-arrays will have shape (1, 1, 1).

CHAPTER

SEVEN

FUNCTIONS

7.1 Retrieval and setting of constants

ATOL	Return or set the value of absolute tolerance for testing numerically tolerant equality.
CHUNKSIZE	Return or set the chunk size for data storage and processing.
FM_THRESHOLD	Return or set the minimum amount of memory to be kept free as a temporary work space.
RTOL	Return or set the default value of relative tolerance for testing numerically tolerant equality.

7.1.1 cf.ATOL

cf.**ATOL**(**atol*)

Return or set the value of absolute tolerance for testing numerically tolerant equality.

Parameters

atol [int, optional] The new value of absolute tolerance.

Returns

out [float or None] If *atol* was not set return the existing value of absolute tolerance, otherwise return *None*.

Examples

```
>>> cf.ATOL()
1e-08
>>> cf.ATOL(1e-10)
>>> cf.ATOL()
1e-10
```

7.1.2 cf.CHUNKSIZE

cf.**CHUNKSIZE**(**new_chunksize*)

Return or set the chunk size for data storage and processing.

When setting the chunk size, the amount of minimum amount of memory to be kept free as a temporary work space is also updated.

Parameters

new_chunksize [int, optional] The new chunk size in bytes.

Returns

out [int or None] If new_chunksize was not set then return the existing chunk size in bytes, otherwise return *None*.

Examples

```
>>> cf.CHUNKSIZE()
104857600
>>> cf.CHUNKSIZE(2**30)
>>> cf.CHUNKSIZE()
1073741824
```

7.1.3 cf.FM_THRESHOLD

`cf.`**`FM_THRESHOLD`**`(*new_minncfm)`

Return or set the minimum amount of memory to be kept free as a temporary work space.

The amount is returned as a number of kibibytes of memory, but set as a number of chunks.

Parameters

new_minncfm [int, optional] The number of chunks to be kept free as a temporary work space.

Returns

out [float or None] If new_minncfm was not set return the existing temporary work space size in kibibytes, otherwise return *None*.

Examples

```
>>> cf.FM_THRESHOLD()
1024000.0
>>> cf.FM_THRESHOLD(20)
>>> cf.FM_THRESHOLD()
2048000.0
```

7.1.4 cf.RTOL

`cf.`**`RTOL`**`(*rtol)`

Return or set the default value of relative tolerance for testing numerically tolerant equality.

Parameters

rtol [int, optional] The new value of relative tolerance.

Returns

out [float or None] If *rtol* was not set return the existing value of relative tolerance, otherwise return *None*.

Examples

```
>>> cf.RTOL()
1.0000000000000001e-05
>>> cf.RTOL(1e-10)
>>> cf.RTOL()
1e-10
```

7.2 Comparison

eq	Return an object for testing whether a variable is equal to the given value.
equals	True if two objects are logically equal, False otherwise.
ge	Return an object for testing whether a variable is greater than or equal to the given value.

Continued on next page

Table 7.2 – continued from previous page

gt	Return an object for testing whether a variable is greater than the given value.
inside	Return an object for testing whether a variable is is inside the given range.
le	Return an object for testing whether a variable is less than or equal to the given value.
lt	Return an object for testing whether a variable is strictly less than a given value.
ne	Return an object for testing whether a variable is not equal to the given value.
outside	Return an object for testing whether a variable is is outside the given range.

7.2.1 cf.eq

cf.**eq**(*value*, *units=None*)

Return an object for testing whether a variable is equal to the given value.

Parameters

value The value which a variable is to be compared with.

Returns

out [Comparison] A Comparison object which will evaluate whether or not the comparison evaluates to True.

Examples

```
>>> c = cf.eq(5)
>>> c
<CF Comparison: x eq 5>
>>> c.evaluate(5)
True
>>> c.evaluate(4)
False
```

7.2.2 cf.equals

cf.**equals**(*x*, *y*, *rtol=None*, *atol=None*, *traceback=False*)

True if two objects are logically equal, False otherwise.

If the first argument, *x* has an *equals* method, then it is used, and in this case `equals(x, y)` is equivalent to `x.equals(y)`.

Parameters

x, y : The objects to compare for equality.

atol [float, optional] The absolute tolerance for all numerical comparisons, By default the value returned by the *ATOL* function is used.

rtol [float, optional] The relative tolerance for all numerical comparisons, By default the value returned by the *RTOL* function is used.

traceback [bool, optional] If True then print a traceback highlighting where the two objects differ.

Returns

out [bool] Whether or not the two objects are equal.

Examples

```
>>> x
<CF Field: rain(10,20)>
>>> cf.equals(x,x)
True
```

```
>>> cf.equals(1.0, 1.0)
True
>>> cf.equals(1.0, 33)
False

>>> cf.equals('a', 'a')
True
>>> cf.equals('a', 'b')
False

>>> type(x), x.dtype
(<type 'numpy.ndarray'>, dtype('int64'))
>>> y=x.copy()
>>> cf.equals(x, y)
True
>>> cf.equals(x, x+1)
False

>>> class A(object): pass
>>> a=A()
>>> b=A()
>>> cf.equals(a, a)
True
>>> cf.equals(a, b)
False
```

7.2.3 cf.ge

`cf.`**`ge`**(*value*, *units=None*)

Return an object for testing whether a variable is greater than or equal to the given value.

Parameters

value The value which a variable is to be compared with.

Returns

out [Comparison] A Comparison object which will evaluate whether or not the comparison evaluates to True.

Examples

```
>>> c = cf.ge(5)
>>> c
<CF Comparison: x ge 5>
>>> c.evaluate(5)
True
>>> c.evaluate(4)
False
```

7.2.4 cf.gt

`cf.`**`gt`**(*value*, *units=None*)

Return an object for testing whether a variable is greater than the given value.

Parameters

value [object] The value which a variable is to be compared with.

Returns

out [Comparison] A Comparison object which will evaluate whether or not the comparison evaluates to True.

Examples

```
>>> c = cf.gt(5)
>>> c
<CF Comparison: x gt 5>
>>> c.evaluate(6)
True
>>> c.evaluate(5)
False
```

7.2.5 cf.inside

`cf.`**`inside`**(*value0*, *value1*, *units=None*)

Return an object for testing whether a variable is is inside the given range.

Parameters

value0 [object] The lower bound of the range which a variable is to be compared with.

value1 [object] The upper bound of the range which a variable is to be compared with.

Returns

out [Comparison] A Comparison object which will evaluate whether or not the comparison evaluates to True.

Examples

```
>>> c = cf.inside(5,7)
>>> c
<CF Comparison: x inside (5, 7)>
>>> c.evaluate(6)
True
>>> c.evaluate(4)
False
```

7.2.6 cf.le

`cf.`**`le`**(*value*, *units=None*)

Return an object for testing whether a variable is less than or equal to the given value.

Parameters

value [object] The value which a variable is to be compared with.

Returns

out [Comparison] A Comparison object which will evaluate whether or not the comparison evaluates to True.

Examples

```
>>> c = cf.le(5)
>>> c
<CF Comparison: x le 5>
>>> c.evaluate(5)
True
>>> c.evaluate(6)
False
```

7.2.7 cf.lt

cf.**lt**(*value*, *units=None*)

Return an object for testing whether a variable is strictly less than a given value.

Parameters

value [object] The value which a variable is to be compared with.

Returns

out [Comparison] A Comparison object which will evaluate whether or not the comparison evaluates to True.

Examples

```
>>> c = cf.lt(5)
>>> c
<CF Comparison: x lt 5>
>>> c.evaluate(4)
True
>>> c.evaluate(5)
False
```

7.2.8 cf.ne

cf.**ne**(*value*, *units=None*)

Return an object for testing whether a variable is not equal to the given value.

Parameters

value [object] The value which a variable is to be compared with.

Returns

out [Comparison] A Comparison object which will evaluate whether or not the comparison evaluates to True.

Examples

```
>>> c = cf.ne(5)
>>> c
<CF Comparison: x ne 5>
>>> c.evaluate(4)
True
>>> c.evaluate(5)
False
```

7.2.9 cf.outside

cf.**outside**(*value0*, *value1*, *units=None*)

Return an object for testing whether a variable is is outside the given range.

Parameters

value0 [object] The lower bound of the range which a variable is to be compared with.

value1 [object] The upper bound of the range which a variable is to be compared with.

Returns

out [Comparison] A Comparison object which will evaluate whether or not the comparison evaluates to True.

Examples

```
>>> c = cf.outside(5)
>>> c
<CF Comparison: x outside (5, 7)>
>>> c.evaluate(4)
True
>>> c.evaluate(6)
False
```

7.3 Input, output and processing

aggregate	Aggregates fields in a field list in place.
dump	Print a description of an object to stdout.
read	Read fields from files from disk or from an OPeNDAP server.
write	Write fields to a CF-netCDF file.

7.3.1 cf.aggregate

cf.**aggregate**(*fields*, *rtol=None*, *atol=None*, *verbose=False*, *messages=False*, *strict_units=False*)

Aggregates fields in a field list in place.

Applies the CF aggregation rules.

Parameters

fields [FieldList] The fields to be aggregated into as few fields as possible.

messages [bool, optional] Print a reason for why a field can't be aggregated.

verbose [bool, optional] Print a summary of the aggregation.

strict_units [bool, optional] If true then assume that variables with the same standard name but missing units are not aggregatable. By default assume that variables with the same standard name but missing units all have equal units

Returns None

Examples

```
>>> f
[<CF Field: eastward_wind(110, 106)>,
 <CF Field: eastward_wind(110, 106)>,
 <CF Field: eastward_wind(110, 106)>,
 <CF Field: air_temperature(12, 73, 96)>,
 <CF Field: eastward_wind(110, 106)>,
 <CF Field: eastward_wind(110, 106)>,
 <CF Field: eastward_wind(1, 1, 110, 106)>,
 <CF Field: air_temperature(96, 73)>]
>>> cf.aggregate(f)
>>> f
[<CF Field: eastward_wind(3, 2, 110, 106)>,
 <CF Field: air_temperature(13, 73, 96)>]
```

7.3.2 cf.dump

cf.**dump**(*x*, ***kwargs*)

Print a description of an object to stdout.

If the object has a *dump* method then this is used to create the output. In this case the arguments are passed to the *dump* method. Otherwise the arguments are ignored and `str(x)` is printed.

Parameters

x [object, optional] The object to print.

kwargs :

Returns None

*8Examples**

```
>>> cf.dump(x)
>>> cf.dump(f, id='field2')
```

7.3.3 cf.read

`cf.read`(*files*, *verbose=False*, *index=None*, *ignore_ioerror=False*, *close=False*, *aggregate=True*, *squeeze=1*, *attr={}*, *priv={}*, *coord={}*, *cellsize={}*)

Read fields from files from disk or from an OPeNDAP server.

Currently supports netCDF and PP formats.

Each field contains it's file name in its *file* attribute and, where appropriate, its netCDF variable name in its *ncvar* attribute. Note that fields may be selected by netCDF variable name by setting a value (or values) of 'ncvar' via the *priv* keyword.

Fields referenced by formula_terms transforms within other fields are not included in the returned list of fields.

Parameters

files [str or sequence of strs] A string or sequence of strings giving the file names or OPeNDAP URLs from which to read fields. For files on disk, the file names may contain wild cards as understood by the python glob module.

index [int, optional] Only return the field with this non-negative index in the otherwise returned full list of fields. By default return all (otherwise selected) fields from the input files.

verbose [bool, optional] If True then print information to stdout.

close [bool, optional] If True then close each file after it has been read.

ignore_ioerror [bool, optional] If True then ignore files which produce IO errors (such as empty files, unknown file formats, etc.).

aggregate [bool or dict, optional] If True or a dictionary then aggregate the fields read in from all input files into as few fields as possible using the CF aggregation rules. If a dictionary then provide these parameters to the aggregate function.

attr [dict, optional] Only return fields matching the given conditions on their standard and non-standard CF attributes. Refer to the field's *match* method for details.

priv [dict, optional] Only return fields matching the given conditions on their attributes other than their standard and non-standard CF attributes. Refer to the field's *match* method for details.

coord [dict, optional] Only return fields matching the given conditions on their coordinates. Refer to the field's *match* method for details.

cellsize [dict, optional] Only return fields matching the given conditions on their coordinates' cellsizes. Refer to the field's *match* method for details.

Returns

out [FieldList] A list of fields.

Raises

IOError : Raised if *ignore_ioerror* is False and there was an I/0 related failure, including unknown file format.

Examples

```
>>> f = cf.read('file*.nc')
>>> type(f)
<class 'cf.field.FieldList'>
>>> f
[<CF Field: pmsl(30, 24)>,
 <CF Field: z-squared(17, 30, 24)>,
 <CF Field: temperature(17, 30, 24)>,
 <CF Field: temperature_wind(17, 29, 24)>]

>>> cf.read('file*.nc')[0:2]
[<CF Field: pmsl(30, 24)>,
 <CF Field: z-squared(17, 30, 24)>]

>>> cf.read('file*.nc', index=0)
[<CF Field: pmsl(30, 24)>]

>>> cf.read('file*.nc')[-1]
<CF Field: temperature_wind(17, 29, 24)>

>>> cf.read('file*.nc', attr={'units': 'K'})
[<CF Field: temperature(17, 30, 24)>,
 <CF Field: temperature_wind(17, 29, 24)>]

>>> cf.read('file*.nc', priv={'ncvar': 'ta'})
[<CF Field: temperature(17, 30, 24)>]

>>> cf.read('file*.nc', attr={'standard_name': '.*pmsl*', 'units':'K|Pa'})[0]
<CF Field: pmsl(30, 24)>

>>> cf.read('file*.nc', attr={'units':['K, 'Pa']})
[<CF Field: pmsl(30, 24)>,
 <CF Field: temperature(17, 30, 24)>,
 <CF Field: temperature_wind(17, 29, 24)>]
```

7.3.4 cf.write

`cf.`**`write`**(*fields*, *file*, *format='NETCDF3_CLASSIC'*, *verbose=False*)

Write fields to a CF-netCDF file.

NetCDF dimension and variable names will be taken from variables' *ncvar* attributes and the space attribute *nc_dimensions* if present, otherwise they are inferred from standard names or set to defaults. NetCDF names may be automatically given a numerical suffix to avoid duplication.

Output netCDF file global attributes are those which occur in the set of CF global attributes and non-standard data variable attributes and which have equal values across all input fields.

Logically identical field components are only written to the file once, apart from when they need to fulfil both dimension coordinate and auxiliary coordinate roles for different data variables.

Parameters

fields [(sequence of) Field or FieldList] The field or fields to write to the file.

file [str] The netCDF file to write fields to.

format [str, optional] The format of the output netCDF file. Valid formats are those supported by the netCDF4 module: 'NETCDF3_CLASSIC',

'NETCDF3_64BIT' 'NETCDF4_CLASSIC' and 'NETCDF4'. The default format is 'NETCDF3_CLASSIC'. Certain write operations may be considerably faster using 'NETCDF4_CLASSIC' or 'NETCDF4'.

verbose [bool, optional] If True then print information to stdout.

Returns None

Raises

ValueError : If a field does not have information required to write certain aspects of a CF-netCDF file.

Examples

```
>>> f
[<CF Field: air_pressure(30, 24)>,
 <CF Field: u_compnt_of_wind(19, 29, 24)>,
 <CF Field: v_compnt_of_wind(19, 29, 24)>,
 <CF Field: potential_temperature(19, 30, 24)>]
>>> write(f, 'file')

>>> type(f)
<class 'cf.field.FieldList'>
>>> type(g)
<class 'cf.field.Field'>
>>> cf.write([f, g], 'file.nc', verbose=True)
[<CF Field: air_pressure(30, 24)>,
 <CF Field: u_compnt_of_wind(19, 29, 24)>,
 <CF Field: v_compnt_of_wind(19, 29, 24)>,
 <CF Field: potential_temperature(19, 30, 24)>]
```

CHAPTER

EIGHT

CLASSES

8.1 Field class

Field	A field construct according to the CF data model.

8.1.1 cf.Field

class cf.**Field**(*attributes={}*)

Bases: cf.variable.Variable

A field construct according to the CF data model.

A field is a container for a data array and metadata comprising properties to describe the physical nature of the data and a coordinate system (called a space) which describes the positions of each element of the data array.

It is structured in exactly the same way as a field construct defined by the CF data model.

The field's space may contain coordinates and cell measures (which themselves contain data arrays and properties to describe them) and transforms to describe how other auxiliary coordinates may be computed.

As in the CF data model, all components of a field are optional.

Initialization

Parameters

attributes [dict, optional] Initialize a new instance with attributes from the dictionary's key/value pairs. Values are deep copied.

CF variable attributes

add_offset	The add_offset CF attribute.
ancillary_variables	
calendar	The calendar CF attribute.
cell_methods	The CellMethods object containing the CF cell methods of the data array.
comment	The comment CF attribute.
Conventions	The Conventions CF attribute.
_FillValue	The _FillValue CF attribute.
flag_masks	The flag_masks CF attribute.
flag_meanings	The flag_meanings CF attribute.
flag_values	The flag_values CF attribute.
history	The history CF attribute.
institution	The institution CF attribute.
leap_month	The leap_month CF attribute.
leap_year	The leap_year CF attribute.

Continued on next page

Table 8.2 – continued from previous page

long_name	The long_name CF attribute.
missing_value	The missing_value CF attribute.
month_lengths	The month_lengths CF attribute.
references	The references CF attribute.
scale_factor	The scale_factor CF attribute.
source	The source CF attribute.
standard_error_multiplier	The standard_error_multiplier CF attribute.
standard_name	The standard_name CF attribute.
title	The title CF attribute.
units	The units CF attribute.
valid_max	The valid_max CF attribute.
valid_min	The valid_min CF attribute.
valid_range	The valid_range CF attribute.

cf.Field.add_offset

Field.**add_offset**

The add_offset CF attribute.

This attribute is only used when writing to a file on disk.

Examples

```
>>> f.add_offset = -4.0
>>> f.add_offset
-4.0
>>> del f.add_offset
```

cf.Field.ancillary_variables

Field.**ancillary_variables**

cf.Field.calendar

Field.**calendar**

The calendar CF attribute.

This attribute is a mirror of the calendar stored in the *Units* attribute.

Examples

```
>>> f.calendar = 'noleap'
>>> f.calendar
'noleap'
>>> del f.calendar
```

cf.Field.cell_methods

Field.**cell_methods**

The CellMethods object containing the CF cell methods of the data array.

Examples

```
>>> f.cell_methods
<CF CellMethods: time: mean (interval: 1.0 month)>
```

cf.Field.comment

Field.**comment**

The comment CF attribute.

Examples

```
>>> f.comment = 'a comment'
>>> f.comment
'a comment'
>>> del f.comment
```

cf.Field.Conventions

Field.**Conventions**

The Conventions CF attribute.

Examples

```
>>> f.Conventions = 'CF-1.5'
>>> f.Conventions
'CF-1.5'
>>> del f.Conventions
```

cf.Field._FillValue

Field.**_FillValue**

The _FillValue CF attribute.

This attribute is forced to be consistent with the *missing_value* attribute as follows:

- Assigning a value to the *missing_value* attribute also assigns the same value to the *_FillValue* attribute whether the latter has been previously set or not.
- Assigning a value to the *_FillValue* attribute also assigns the same value to the *missing_value* attribute, but only if the latter has previously been defined.

Examples

```
>>> f._FillValue = -1e30
```

cf.Field.flag_masks

Field.**flag_masks**

The flag_masks CF attribute.

Stored as a 1-d numpy array but may be set as array-like object.

Examples

```
>>> f.flag_masks = numpy.array([1, 2, 4], dtype='int8')
>>> f.flag_masks
array([1, 2, 4], dtype=int8)
>>> f.flag_masks = 1
>>> f.flag_masks
array([1])
```

cf.Field.flag_meanings

Field.**flag_meanings**
The flag_meanings CF attribute.

Stored as a 1-d numpy string array but may be set as a space delimited string or any array-like object.

Examples

```
>>> f.flag_meanings = 'low medium      high'
>>> f.flag_meanings
array(['low', 'medium', 'high'],
      dtype='|S6')
>>> f.flag_meanings = ['left', 'right']
>>> f.flag_meanings
array(['left', 'right'],
      dtype='|S5')
>>> f.flag_meanings = 'ok'
>>> f.flag_meanings
array(['ok'],
      dtype='|S2')
>>> f.flag_meanings = numpy.array(['a', 'b'])
>>> f.flag_meanings
array(['a', 'b'],
      dtype='|S1')
```

cf.Field.flag_values

Field.**flag_values**
The flag_values CF attribute.

Stored as a 1-d numpy array but may be set as any array-like object.

Examples

```
>>> f.flag_values = ['a', 'b', 'c']
>>> f.flag_values
array(['a', 'b', 'c'], dtype='|S1')
>>> f.flag_values = numpy.arange(4)
>>> f.flag_values
array([1, 2, 3, 4])
>>> f.flag_values = 1
>>> f.flag_values
array([1])
```

cf.Field.history

Field.**history**
The history CF attribute.

Examples

```
>>> f.history = 'created on 2012/10/01'
>>> f.history
'created on 2012/10/01'
>>> del f.history
```

cf.Field.institution

Field.**institution**
The institution CF attribute.

Examples

```
>>> f.institution = 'University of Reading'
>>> f.institution
'University of Reading'
>>> del f.institution
```

cf.Field.leap_month

Field.**leap_month**
The leap_month CF attribute.

Examples

```
>>> f.leap_month = 2
>>> f.leap_month
2
>>> del f.leap_month
```

cf.Field.leap_year

Field.**leap_year**
The leap_year CF attribute.

Examples

```
>>> f.leap_year = 1984
>>> f.leap_year
1984
>>> del f.leap_year
```

cf.Field.long_name

Field.**long_name**
The long_name CF attribute.

Examples

```
>>> f.long_name = 'zonal_wind'
>>> f.long_name
'zonal_wind'
>>> del f.long_name
```

cf.Field.missing_value

Field.**missing_value**
The missing_value CF attribute.

This attribute is forced to be consistent with the *_FillValue* attribute as follows:

- Assigning a value to the *missing_value* attribute also assigns the same value to the *_FillValue* attribute whether the latter has been previously set or not.
- Assigning a value to the *_FillValue* attribute also assigns the same value to the *missing_value* attribute, but only if the latter has previously been defined.

Examples

```
>>> f.missing_value = 1e30
>>> f.missing_value
1e30
>>> f._FillValue
1e30
>>> del f.missing_value
>>> f._FillValue
1e30
```

cf.Field.month_lengths

`Field.`**`month_lengths`**
The month_lengths CF attribute.

cf.Field.references

`Field.`**`references`**
The references CF attribute.

Examples

```
>>> f.references = 'some references'
>>> f.references
'some references'
>>> del f.references
```

cf.Field.scale_factor

`Field.`**`scale_factor`**
The scale_factor CF attribute.

This attribute is only used when writing to a file on disk.

Examples

```
>>> f.scale_factor = 10.0
>>> f.scale_factor
10.0
>>> del f.scale_factor
```

cf.Field.source

`Field.`**`source`**
The source CF attribute.

Examples

```
>>> f.source = 'radiosonde'
>>> f.source
'radiosonde'
>>> del f.source
```

cf.Field.standard_error_multiplier

`Field.`**`standard_error_multiplier`**
The standard_error_multiplier CF attribute.

Examples

```
>>> f.standard_error_multiplier = 2.0
>>> f.standard_error_multiplier
2.0
>>> del f.standard_error_multiplier
```

cf.Field.standard_name

Field.**standard_name**
The standard_name CF attribute.

Examples

```
>>> f.standard_name = 'time'
>>> f.standard_name
'time'
>>> del f.standard_name
```

cf.Field.title

Field.**title**
The title CF attribute.

Examples

```
>>> f.title = 'model data'
>>> f.title
'model data'
>>> del f.title
```

cf.Field.units

Field.**units**
The units CF attribute.

This attribute is a mirror of the units stored in the *Units* attribute.

Examples

```
>>> f.units = 'K'
>>> f.units
'K'
>>> del f.units
```

cf.Field.valid_max

Field.**valid_max**
The valid_max CF attribute.

Examples

```
>>> f.valid_max = 100.0
>>> f.valid_max
100.0
>>> del f.valid_max
```

cf.Field.valid_min

Field.**valid_min**
The valid_min CF attribute.

Examples

```
>>> f.valid_min = 100.0
>>> f.valid_min
100.0
>>> del f.valid_min
```

cf.Field.valid_range

Field.**valid_range**
The valid_range CF attribute.

May be set as a numpy array, a list or a tuple. Always returned as a tuple.

Examples

```
>>> f.valid_range = array([100., 400.])
>>> f.valid_range
(100.0, 400.0)
>>> del f.valid_range
>>> f.valid_range = (50., 450.)
```

Data attributes

array	A numpy array deep copy of the data array.
Data	The *Data* object containing the data array.
dtype	Numpy data-type of the data array.
_FillValue	The _FillValue CF attribute.
first_datum	The first element of the data array.
is_scalar	True if and only if the data array is a scalar array.
last_datum	The last element of the data array.
mask	The boolean missing data mask of the data array.
ndim	Number of dimensions in the data array.
shape	Tuple of the data array's dimension sizes.
size	Number of elements in the data array.
subset	Return a new field whose data and space are subsetted in a consistent manner.
Units	The *Units* object containing the units of the data array.
varray	A numpy array view of the data array.

cf.Field.array

Field.**array**
A numpy array deep copy of the data array.

Changing the returned numpy array does not change the data array.

Examples

```
>>> a = f.array
>>> type(a)
<type 'numpy.ndarray'>
>>> a
array([0, 1, 2, 3, 4])
>>> a[0] = 999
>>> f.array
array([0, 1, 2, 3, 4])
```

cf.Field.Data

Field.**Data**

The *Data* object containing the data array.

Examples

```
>>> f.Data
<CF Data: >
```

cf.Field.dtype

Field.**dtype**

Numpy data-type of the data array.

Examples

```
>>> f.dtype
dtype('float64')
```

cf.Field._FillValue

Field.**_FillValue**

The _FillValue CF attribute.

This attribute is forced to be consistent with the *missing_value* attribute as follows:

- Assigning a value to the *missing_value* attribute also assigns the same value to the *_FillValue* attribute whether the latter has been previously set or not.
- Assigning a value to the *_FillValue* attribute also assigns the same value to the *missing_value* attribute, but only if the latter has previously been defined.

Examples

```
>>> f._FillValue = -1e30
```

cf.Field.first_datum

Field.**first_datum**

The first element of the data array.

Equivalent to `f.subset[(0,)*f.ndim].array.item()`

Examples

```
>>> f.array
array([[1, 2],
       [3, 4]])
>>> f.first_datum
1
```

cf.Field.is_scalar

Field.**is_scalar**

True if and only if the data array is a scalar array.

Examples

```
>>> f.array
array(2)
>>> f.is_scalar
True

>>> f.array
array([2])
>>> f.is_scalar
False
```

cf.Field.last_datum

`Field.`**`last_datum`**

The last element of the data array.

Equivalent to `f.subset[(-1,)*f.ndim].array.item()`

Examples

```
>>> f.array
array([[1, 2],
       [3, 4]])
>>> f.first_datum
4
```

cf.Field.mask

`Field.`**`mask`**

The boolean missing data mask of the data array.

Returned as a *Data* object. The mask may be set to 'no missing data' by deleting the attribute.

Examples

```
>>> f.shape
(12, 73, 96)
>>> m = f.mask
>>> type(m)
<cf.data.Data>
>>> m.dtype
dtype('bool')
>>> m.shape
[12, 73, 96]
>>> m.array.shape
(12, 73, 96)

>>> del f.mask
>>> f.array.mask
False
>>> import numpy
>>> f.array.mask is numpy.ma.nomask
True
```

cf.Field.ndim

`Field.`**`ndim`**

Number of dimensions in the data array.

Examples

```
>>> f.shape
(73, 96)
>>> f.ndim
2
```

cf.Field.shape

`Field.`**`shape`**

Tuple of the data array's dimension sizes.

Examples

```
>>> f.shape
(73, 96)
```

cf.Field.size

`Field.`**`size`**

Number of elements in the data array.

Examples

```
>>> f.shape
(73, 96)
>>> f.size
7008
```

cf.Field.subset

`Field.`**`subset`**

Return a new field whose data and space are subsetted in a consistent manner.

This attribute may be indexed to select a subset from dimension index values (square brackets: `f.subset[indices]`) or called to select a subset from dimension coordinate array values (round brackets: `f.subset(coordinate_values)`)

Subsetting by indexing

Subsetting by dimension indices uses an extended Python slicing syntax, which is similar numpy array indexing. There are two extensions to the numpy indexing functionality:

- Size 1 dimensions are never removed.

  An integer index i takes the i-th element but does not reduce the rank of the output array by one.

- When advanced indexing is used on more than one dimension, the advanced indices work independently.

  When more than one dimension's slice is a 1-d boolean array or 1-d sequence of integers, then these indices work independently along each dimension (similar to the way vector subscripts work in Fortran), rather than by their elements.

Subsetting by coordinate values

Subsetting by coordinate values allows a subsetted field to be defined by particular coordinate values of its space.

Subsetting by coordinate values is functionally equivalent to subsetting by indexing - internally, the selected coordinate values are in fact converted to dimension indices.

Coordinate values are provided as arguments to a call to the subset method.

The benefits to subsetting in this fashion are:

- The dimensions to be subsetted are identified by name.
- The position in the data array of each dimension need not be known.
- Dimensions for which no subsetting is required need not be specified.
- Size 1 dimensions of the space which are not spanned by the data array may be specified.

Examples

cf.Field.Units

Field.**Units**

The *Units* object containing the units of the data array.

Stores the units and calendar CF attributes in an internally consistent manner. These attributes are mirrored by the *units* and *calendar* attributes respectively.

Examples

```
>>> f.Units
<CF Units: K>
```

cf.Field.varray

Field.**varray**

A numpy array view of the data array.

Changing the elements of the returned view changes the data array.

Examples

```
>>> a = f.varray
>>> type(a)
<type 'numpy.ndarray'>
>>> a
array([0, 1, 2, 3, 4])
>>> a[0] = 999
>>> f.varray
array([999, 1, 2, 3, 4])
```

Other attributes

attributes	A dictionary of the standard and non-standard CF attributes.
Flags	A Flags object containing self-describing CF flag values.
space	The Space object containing the coordinate system (space).

cf.Field.attributes

Field.**attributes**

A dictionary of the standard and non-standard CF attributes.

Note that modifying the returned dictionary will not change the attributes.

Examples

```
>>> f.attributes
{'Conventions': 'CF-1.0',
 '_FillValue': 1e+20,
 'cell_methods': <CF CellMethods: time: mean>,
 'experiment_id': 'stabilization experiment (SRES A1B)',
 'long_name': 'Surface Air Temperature',
 'standard_name': 'AIR_TEMP',
```

```
 'title': 'SRES A1B',
 'units': 'K'}
```

cf.Field.Flags

Field.**Flags**
A Flags object containing self-describing CF flag values.

Stores the flag_values, flag_meanings and flag_masks CF attributes in an internally consistent manner.

Examples

```
>>> f.Flags
<CF Flags: flag_values=[0 1 2], flag_masks=[0 2 2], flag_meanings=['low' 'medium' 'high']
```

cf.Field.space

Field.**space**
The Space object containing the coordinate system (space).

Examples

```
>>> f.space
<CF Space: (12, 19, 73, 96)>
```

Methods

binary_mask	return new variable containing binary mask
chunk	Partition the field in place for LAMA functionality.
coord	Return a coordinate of the field.
copy	Return a deep copy.
delattr	Delete a standard or non-standard CF attribute.
dump	Return a string containing a full description of the instance.
equals	True if two fields are logically equal, False otherwise.
expand_dims	Expand the shape of the data array in place.
extract	Return the instance if it matches the given conditions.
getattr	Get a standard or non-standard CF attribute.
finalize	Finalize a newly created field.
hasattr	Return True if the variable has standard or non-standard CF attribute.
match	Determine whether or not a field satisfies conditions.
name	Return a name for the variable.
reverse_dims	Reverse the directions of data array axes in place.
setattr	Set a standard or non-standard CF attribute.
squeeze	Remove size 1 dimensions from the field's data array in place.
transpose	Permute the dimensions of the data array in place.
unsqueeze	Insert size 1 dimensions from the field's space into its data array in place.

cf.Field.binary_mask

Field.**binary_mask**()
return new variable containing binary mask

cf.Field.chunk

Field.**chunk**(*chunksize=None*, *extra_boundaries=None*, *chunk_dims=None*)
Partition the field in place for LAMA functionality.

Parameters chunksize : int, optional

extra_boundaries : sequence of lists or tuples, optional

chunk_dims : sequence of lists or tuples, optional

Returns extra_boundaries, chunk_dims : {list, list}

cf.Field.coord

`Field.coord`(*arg*, *role=None*, *key=False*, *exact=False*, *maximal_match=True*)

Return a coordinate of the field.

Note that the returned coordinate is an object identity to the coordinate stored in the space so, for example, a coordinate's attributes may be changed in-place:

```
>>> f.coord('height').long_name
AttributeError: 'Coordinate' object has no attribute 'long_name'
>>> f.coord('hei').long_name = 'HEIGHT'
>>> f.coord('heigh').long_name
'HEIGHT'
```

Or a deep copy may be made:

```
>>> c = f.coord('height').copy()
```

Parameters

arg [str or dict or int] The identify of the coordinate. One of:

- A string containing (an abbreviation) of its standard name (such as 'time').
- A string containing its identifier in the field's space (such as 'dim2').
- A dictionary containing one or more (abbreviations of) attributes names and their values as its key/value pairs (such as {'long_name': 'something'}). If two or more attributes are specified then the returned coordinate must satisfy all criteria, unless *maximal_match* is False in which case the returned coordinate will satisfy all criteria at least one of the criteria.
- An integer given the position of a dimension in the field's data array.

exact [bool, optional] If True then assume that the value of a name given by *arg* is exact. By default it is considered to be an abbreviation.

key [bool, optional] Return the field's space's identifier for the coordinate.

role [bool, optional] Restrict the search to coordinates of the given role. Valid values are 'dim' and 'aux', for dimension and auxiliary coordinate constructs. By default both types are considered.

Returns

out [Coordinate or str or None] The requested coordinate, or the field's space's identifier for the coordinate, or *None* if no coordinate could be found.

Examples

```
>>> f.coord('lon')
<CF Coordinate: longitude(128)>
>>> f.coord('lon', key=True)
'dim2'
>>> f.coord('lonX', key=True)
None
>>> f.coord('lon', exact=True)
None
```

```
>>> f.coord('longitude', exact=True)
<CF Coordinate: longitude(128)>
```

cf.Field.copy

Field.**copy**()

Return a deep copy.

Equivalent to `copy.deepcopy(f)`

Returns

out : The deep copy.

Examples

```
>>> g = f.copy()
```

cf.Field.delattr

Field.**delattr**(*attr*)

Delete a standard or non-standard CF attribute.

Parameters

attr [str] The name of the attribute to delete.

Returns None

Raises

AttributeError : If the variable does not have the named attribute.

Examples

```
>>> f.delattr('standard_name')
>>> f.delattr('foo')
```

cf.Field.dump

Field.**dump**(*id=None*)

Return a string containing a full description of the instance.

Parameters

id [str, optional] Set the common prefix of component names. By default the instance's class name is used.

Returns

out [str] A string containing the description.

Examples

```
>>> x = f.dump()
>>> print f.dump()
>>> print f.dump(id='pressure_field')
```

cf.Field.equals

Field.**equals**(*other*, *rtol=None*, *atol=None*, *traceback=False*)

True if two fields are logically equal, False otherwise.

The *Conventions* attribute is ignored in the comparison.

Parameters

other : The object to compare for equality.

atol [float, optional] The absolute tolerance for all numerical comparisons, By default the value returned by the *ATOL* function is used.

rtol [float, optional] The relative tolerance for all numerical comparisons, By default the value returned by the *RTOL* function is used.

traceback [bool, optional] If True then print a traceback highlighting where the two instances differ.

Returns

out [bool] Whether or not the two instances are equal.

Examples

```
>>> f.Conventions
'CF-1.0'
>>> g = f.copy()
>>> g.Conventions = 'CF-1.5'
>>> f.equals(g)
True
```

In this example, two fields differ only by the long name of their time coordinates. The traceback shows that they differ in their spaces, that they differ in their time coordinates and that the long name has not been matched.

```
>>> g = f.copy()
>>> g.coord('time').long_name = 'something_else'
>>> f.equals(g, traceback=True)
Coordinate: Different long_name: 'time', 'something else'
Coordinate: Different long_name: 'time', 'latitude in rotated pole grid'
Coordinate: Different long_name: 'time', 'longitude in rotated pole grid'
Space: Different coordinate: <CF Coordinate: time(12)>
Field: Different 'space': <CF Space: (73, 96, 12)>, <CF Space: (73, 96, 12)>
False
```

cf.Field.expand_dims

Field.**expand_dims**(*arg=None*, *axis=0*)

Expand the shape of the data array in place.

Insert a new size 1 axis, corresponding to a given position in the data array shape.

Parameters

arg [str, optional] The dimension to insert. if specified, the dimension should exist in the field's space and is identified by its standard name or by the space's internal dimension name. By default, insert a new dimension which doesn't yet exist in the field's space.

axis [int, optional] Position (amongst axes) where new axis is to be inserted. By default, insert at position 0.

Returns None

Examples

```
>>> f.expand_dims()
>>> f.expand_dims(axis=1)
>>> f.expand_dims('height')
>>> f.expand_dims('height', 3)
>>> f.expand_dims('dim1', 3)
```

cf.Field.extract

`Field.`**`extract`**(**args*, ***kwargs*)

Return the instance if it matches the given conditions.

Equivalent to:

```
def extract(f, *args, **kwargs):
    if f.match(*args, **kwargs):
        return f
    raise ValueError('')
```

Parameters

args, kwargs : As for the *match* method.

Returns

out : The variable as an object identity, if it matches the given conditions.

Raises

ValueError : If the variable does not match the conditions.

cf.Field.getattr

`Field.`**`getattr`**(*attr*, **default*)

Get a standard or non-standard CF attribute.

Parameters

attr [str] The name of the attribute to get.

default [optional] Return *default* if and only if the variable does not have the named attribute.

Returns

out : The value of the named attribute, or the default value.

Raises

AttributeError : If the variable does not have the named attribute a default value has not been set.

Examples

```
>>> f.getattr('standard_name')
>>> f.getattr('standard_name', None)
>>> f.getattr('foo')
```

cf.Field.finalize

`Field.`**`finalize`**(*chunksize=None*)

Finalize a newly created field.

It is essential that this is carried out on every new field to ensure that methods and functions of fields work correctly. Finalization entails:

•Expanding scalar coordinate and cell measures to 1-d.

•Setting the direction of the space's dimensions.

•Conforming the space's internal dimension names.

•Partitioning the data arrays for LAMA functionality.

Parameters

chunksize [int, optional] Set the maximum size in bytes of sub-arrays for data storage and processing. By default the size used by the *CHUNKSIZE* function is used.

Returns None

Examples

```
>>> f.finalize()
>>> f.finalize(2**30)
```

cf.Field.hasattr

`Field.`**`hasattr`**`(`*attr*`)`

Return True if the variable has standard or non-standard CF attribute.

Parameters

attr [str] The name of the attribute.

Returns

out [bool] True if the instance has the attribute.

Examples

```
>>> f.hasattr('standard_name')
>>> f.hasattr('foo')
```

cf.Field.match

`Field.`**`match`**`(`*attr={}*, *priv={}*, *coord={}*, *cellsize={}*`)`

Determine whether or not a field satisfies conditions.

Conditions may be specified on the field's attributes, coordinate values and coordinate cell sizes.

Parameters

attr [dict, optional] Dictionary for which each key/value pair is a standard or non-standard CF attribute name and a condition for the attribute to be tested against. If the value is a sequence of conditions then the attribute matches if at least one of the conditions is passed.

In general, a condition is any object and it is passed if the attribute is equal to the object, with the following exception:

- If the attribute is string-valued, then the condition may be a regular expression pattern recognised by the *re* module and the condition is passed if the attribute matches the regular expression. Special characters for the start and end of the string are assumed and need not be included. For example, '.*wind' is equivalent to '^.*wind$'.

priv [dict, optional] Dictionary for which each key/value pair is an attribute name other than their standard and non-standard CF attribute and a condition for the attribute to be tested against. If the value is a sequence of conditions then the attribute matches if at least one of the conditions is passed.

In general, a condition behaves as for *attr*, with the following exception:

- For the *Units* attribute, the condition is passed if the attribute is equivalent (rather than equal) to the object. (Note that this behaviour does not apply to the *units* attribute.)

coord [dict, optional] Dictionary for which each key/value pair is a coordinate name and a condition for the coordinate to be tested against. A match occurs if at least one element of the coordinate's array passes the condition. If the value is a sequence of conditions then the coordinate matches if at least one of the conditions is passed.

cellsize [dict, optional] Dictionary for which each key/value pair is a coordinate name and a condition for the coordinate's cell sizes to be tested against. A match occurs if all of the coordinate's cell's sizes pass the condition. Only one cell size condition may be given per coordinate.

If the same coordinate is specified in the *coord* and *cellsize* dictionaries then the cell size condition only needs to be passed for cells which pass all if the coordinate's conditions.

Returns

out [bool] True if the field satisfies the given criteria, False otherwise.

Examples

```
>>> print f
Data             : air_temperature(time, latitude, longitude)
Cell methods     : time: mean
Dimensions       : time(12) = [15, ..., 345] days since 1860-1-1
                 : latitude(73) = [-90, ..., 90] degrees_north
                 : longitude(96) = [0, ..., 356.25] degrees_east
                 : height(1) = [2] m

>>> f.match(attr={'standard_name': 'air_temperature'})
True
>>> f.match(attr={'standard_name': ['air_temperature']})
True
>>> f.match(attr={'standard_name': ['air_temperature', 'air_pressure']})
True
>>> f.match(attr={'standard_name': '.*temperature.*'})
True
>>> f.match(attr={'standard_name': ['.*temperature.*', 'air_pressure']})
True
>>> f.match(attr={'standard_name': '.*pressure.*'})
False

>>> f.match(attr={'Units': 'K'})
True
>>> f.match(attr={'Units': cf.Units('1.8 K @ 459.67')})
True
>>> f.match(attr={'Units': [cf.Units('Pa'), 'K']})
True
>>> f.match(attr={'Units': cf.Units('Pa')})
False

>>> f.match(attr={'cell_methods': 'time: mean'})
True
>>> f.match(attr={'cell_methods': cf.CellMethods('time: mean')})
True
>>> f.match(attr={'cell_methods': ['time: mean', 'time: max']})
True
>>> f.match(attr={'cell_methods': cf.CellMethods('time: max')})
False
>>> f.match(attr={'cell_methods': 'time: mean time: min')})
False
```

```
>>> f.match(coord={'latitude': 0})
False
>>> f.match(coord={'latitude': [0, cf.gt(30)]})
True

>>> f.match(cellsize={'time': cf.inside(28, 31, 'days')})
True
```

cf.Field.name

`Field.`**`name`**(*long_name=False*, *ncvar=False*, *default=None*)

Return a name for the variable.

Returns the standard_name, long_name (if requested) or netCDF variable name (if requested), whichever it finds first, otherwise returns a default name.

Parameters long_name : bool, optional

If True, return the long_name if standard_name does not exist.

ncvar : bool, optional

If True, return ncvar if neither the standard_name nor long_name has already been returned.

default : str, optional

Return default if neither standard_name, long_name nor ncvar has already been returned.

Returns

name [str] The name of the variable.

cf.Field.reverse_dims

`Field.`**`reverse_dims`**(*axes=None*)

Reverse the directions of data array axes in place.

Equivalent to indexing the specified dimensions with `::-1`.

Parameters

axes [sequence, optional] The dimensions to have reversed directions. By default, reverse the directions of all dimensions. Dimensions for direction reversal may be identified with one of, or a sequence of any combination of:

- A dimension's standard name.
- The integer position of a dimension in the data array.
- The field's space's internal name of a dimension

Returns None

Examples

```
>>> f.reverse_dims()
>>> f.reverse_dims('time')
>>> f.reverse_dims(1)
>>> f.reverse_dims('dim2')
>>> f.reverse_dims(['time', 1, 'dim2'])
```

cf.Field.setattr

Field.**setattr**(*attr*, *value*)
Set a standard or non-standard CF attribute.

Parameters

attr [str] The name of the attribute to set.

value : The value for the attribute.

Returns None

Examples

```
>>> f.setattr('standard_name', 'time')
>>> f.setattr('foo', 12.5)
```

cf.Field.squeeze

Field.**squeeze**(*axes=None*)
Remove size 1 dimensions from the field's data array in place.

Parameters

axes [optional] The size 1 axes to remove. By default, all size 1 axes are removed. Size 1 axes for removal may be identified with one of, or a sequence of any combination of:

- A dimension's standard name.
- The integer position of a dimension in the data array.
- The field's space's internal name of a dimension

Returns None

Examples

```
>>> f.squeeze()
>>> f.squeeze('time')
>>> f.squeeze(1)
>>> f.squeeze('dim2')
>>> f.squeeze([1, 'time', 'dim2'])
```

cf.Field.transpose

Field.**transpose**(*axes=None*)
Permute the dimensions of the data array in place.

Parameters

axes [sequence, optional] The new order of the data array. By default, reverse the dimensions' order, otherwise the axes are permuted according to the values given. The values of the sequence may be any combination of:

- A dimension's standard name.
- The integer position of a dimension in the data array.
- The field's space's internal name of a dimension

Returns None

Examples

```
>>> f.transpose()
>>> f.transpose(['latitude', 'time', 'longitude'])
>>> f.transpose([1, 0, 2])
>>> f.transpose((1, 'time', 'dim2'))
```

cf.Field.unsqueeze

`Field.`**`unsqueeze`**`()`

Insert size 1 dimensions from the field's space into its data array in place.

Returns None

Examples

```
>>> f.unsqueeze()
```

8.2 Field component classes

`CellMeasures`	A CF cell measures construct containing information that is needed
`CellMethods`	A CF cell methods object to describe the characteristic of a field
`Coordinate`	A CF dimension or auxiliary coordinate construct.
`CoordinateBounds`	A CF coordinate's bounds object containing cell boundaries or
`Data`	An N-dimensional data array with units and masked values.
`Flags`	Self-describing CF flag values.
`Space`	Completely describe a field's coordinate system (space).
`Transform`	A CF transform construct.
`Units`	Store, combine and compare physical units and convert numeric values to different units.

8.2.1 cf.CellMeasures

class `cf.`**`CellMeasures`**(*attributes={}*)

Bases: `cf.variable.Variable`

A CF cell measures construct containing information that is needed about the size, shape or location of the field's cells.

It is a variable which contains a data array and metadata comprising properties to describe the physical nature of the data.

The name of spatial measure being represented is stored in the *measure* attribute.

Initialization

Parameters

attributes [dict, optional] Initialize a new instance with attributes from the dictionary's key/value pairs. Values are deep copied.

`binary_mask`()

return new variable containing binary mask

`chunk`(*chunksize=None*, *extra_boundaries=[]*, *adim=[]*)

Partition the data array using Large Amounts of Massive Arrays (LAMA) functionality.

Parameters chunksize : int

extra_boundaries : list

adim : list

Returns extra_boundaries, adim : {list, list}

copy(*_omit_special=()*)
Return a deep copy.

Equivalent to `copy.deepcopy(f)`.

Returns

out : The deep copy.

Examples

```
>>> g = f.copy()
```

delattr(*attr*)
Delete a standard or non-standard CF attribute.

Parameters

attr [str] The name of the attribute to delete.

Returns None

Raises

AttributeError : If the variable does not have the named attribute.

Examples

```
>>> f.delattr('standard_name')
>>> f.delattr('foo')
```

dump(*id=None*)
Return a string containing a full description of the cell measure.

Parameters

id: str, optional Set the common prefix of variable component names. By default the instance's class name is used.

Returns

out [str] A string containing the description.

Examples

```
>>> x = c.dump()
>>> print c.dump()
>>> print c.dump(id='area_measure')
```

equals(*other*, *rtol=None*, *atol=None*, *traceback=False*, *ignore=set([])*)
True if two variables are logically equal, False otherwise.

Parameters

other [] The object to compare for equality.

atol [float, optional] The absolute tolerance for all numerical comparisons, By default the value returned by the *ATOL* function is used.

rtol [float, optional] The relative tolerance for all numerical comparisons, By default the value returned by the *RTOL* function is used.

traceback [bool, optional] If True then print a traceback highlighting where the two instances differ.

ignore [set, optional] Omit these attributes from the comparison.

Returns

out [bool] Whether or not the two instances are equal.

Examples

```
>>> f.equals(f)
True
>>> g = f + 1
>>> f.equals(g)
False
>>> g -= 1
>>> f.equals(g)
True
>>> f.setattr('name', 'name0')
>>> g.setattr('name', 'name1')
>>> f.equals(g)
False
```

expand_dims(*axis*, *dim*, *direction*)
axis is an integer dim is a string direction is a boolean

extract(**args*, ***kwargs*)
Return the instance if it matches the given conditions.

Equivalent to:

```
def extract(f, *args, **kwargs):
    if f.match(*args, **kwargs):
        return f
    raise ValueError('')
```

Parameters

args, kwargs : As for the *match* method.

Returns

out : The variable as an object identity, if it matches the given conditions.

Raises

ValueError : If the variable does not match the conditions.

getattr(*attr*, **default*)
Get a standard or non-standard CF attribute.

Parameters

attr [str] The name of the attribute to get.

default [optional] Return *default* if and only if the variable does not have the named attribute.

Returns

out : The value of the named attribute, or the default value.

Raises

AttributeError : If the variable does not have the named attribute a default value has not been set.

Examples

```
>>> f.getattr('standard_name')
>>> f.getattr('standard_name', None)
>>> f.getattr('foo')
```

hasattr(*attr*)
Return True if the variable has standard or non-standard CF attribute.

Parameters

attr [str] The name of the attribute.

Returns

out [bool] True if the instance has the attribute.

Examples

```
>>> f.hasattr('standard_name')
>>> f.hasattr('foo')
```

match(*attr={}*, *priv={}*)

Determine whether or not a variable matches conditions on its attributes.

Parameters

attr [dict] Dictionary for which each key/value pair is a standard or non-standard CF attribute name and a condition for the attribute to be tested against. If the value is a sequence of conditions then the attribute matches if at least one of the conditions is passed.

In general, a condition is any object and it is passed if the attribute is equal to the object, with the following exceptions:

- If the attribute is string-valued, then the condition may be a regular expression pattern recognised by the *re* module and the condition is passed if the attribute matches the regular expression. Special characters for the start and end of the string are assumed and need not be included. For example, '.*wind' is equivalent to '^.*wind$'.
- For the 'Units' attribute, the condition is passed if the attribute is equivalent (rather than equal) to the object. (Note that this behaviour does not apply to the 'units' attribute.)

priv [dict, optional] Dictionary for which each key/value pair is an attribute name other than their standard and non-standard CF attribute and a condition for the attribute to be tested against. If the value is a sequence of conditions then the attribute matches if at least one of the conditions is passed.

In general, a condition behaves as for *attr*, with the following exception:

- For the *Units* attribute, the condition is passed if the attribute is equivalent (rather than equal) to the object. (Note that this behaviour does not apply to the *units* attribute.)

Returns

out [bool] Whether or not the variable matches the given criteria.

Examples

name(*long_name=None*, *ncvar=None*, *default=None*)

reverse_dims(*axes=None*)

Reverse the directions of the data array dimensions in place.

Equivalent to indexing the specified dimensions with `::-1`.

Parameters

axes [int or sequence of ints] By default all dimensions are reversed.

Returns

out [list of ints] The axes which were reversed, in arbitrary order.

Examples

```
>>> f.reverse_dims()
>>> f.reverse_dims(1)
>>> f.reverse_dims([0, 1])
```

setattr(*attr*, *value*)
Set a standard or non-standard CF attribute.

Parameters

attr [str] The name of the attribute to set.

value : The value for the attribute.

Returns None

Examples

```
>>> f.setattr('standard_name', 'time')
>>> f.setattr('foo', 12.5)
```

squeeze(*axes=None*)
axes are integers

transpose(*axes=None*)
axes are integers

Data
The *Data* object containing the data array.

Examples

```
>>> f.Data
<CF Data: >
```

Units
The *Units* object containing the units of the data array.

Stores the units and calendar CF attributes in an internally consistent manner. These attributes are mirrored by the *units* and *calendar* attributes respectively.

Examples

```
>>> f.Units
<CF Units: K>
```

add_offset
The add_offset CF attribute.

This attribute is only used when writing to a file on disk.

Examples

```
>>> f.add_offset = -4.0
>>> f.add_offset
-4.0
>>> del f.add_offset
```

array
A numpy array deep copy of the data array.

Changing the returned numpy array does not change the data array.

Examples

```
>>> a = f.array
>>> type(a)
<type 'numpy.ndarray'>
>>> a
array([0, 1, 2, 3, 4])
>>> a[0] = 999
>>> f.array
array([0, 1, 2, 3, 4])
```

attributes

A dictionary of the standard and non-standard CF attributes.

Note that modifying the returned dictionary will not change the attributes.

Examples

```
>>> f.attributes
{'Conventions': 'CF-1.0',
 '_FillValue': 1e+20,
 'cell_methods': <CF CellMethods: time: mean>,
 'experiment_id': 'stabilization experiment (SRES A1B)',
 'long_name': 'Surface Air Temperature',
 'standard_name': 'AIR_TEMP',
 'title': 'SRES A1B',
 'units': 'K'}
```

calendar

The calendar CF attribute.

This attribute is a mirror of the calendar stored in the *Units* attribute.

Examples

```
>>> f.calendar = 'noleap'
>>> f.calendar
'noleap'
>>> del f.calendar
```

comment

The comment CF attribute.

Examples

```
>>> f.comment = 'a comment'
>>> f.comment
'a comment'
>>> del f.comment
```

dtype

Numpy data-type of the data array.

Examples

```
>>> f.dtype
dtype('float64')
```

first_datum

The first element of the data array.

Equivalent to `f.subset[(0,)*f.ndim].array.item()`

Examples

```
>>> f.array
array([[1, 2],
       [3, 4]])
>>> f.first_datum
1
```

history

The history CF attribute.

Examples

```
>>> f.history = 'created on 2012/10/01'
>>> f.history
```

```
'created on 2012/10/01'
>>> del f.history
```

is_scalar

True if and only if the data array is a scalar array.

Examples

```
>>> f.array
array(2)
>>> f.is_scalar
True

>>> f.array
array([2])
>>> f.is_scalar
False
```

last_datum

The last element of the data array.

Equivalent to `f.subset[(-1,)*f.ndim].array.item()`

Examples

```
>>> f.array
array([[1, 2],
       [3, 4]])
>>> f.first_datum
4
```

leap_month

The leap_month CF attribute.

Examples

```
>>> f.leap_month = 2
>>> f.leap_month
2
>>> del f.leap_month
```

leap_year

The leap_year CF attribute.

Examples

```
>>> f.leap_year = 1984
>>> f.leap_year
1984
>>> del f.leap_year
```

long_name

The long_name CF attribute.

Examples

```
>>> f.long_name = 'zonal_wind'
>>> f.long_name
'zonal_wind'
>>> del f.long_name
```

mask

The boolean missing data mask of the data array.

Returned as a *Data* object. The mask may be set to 'no missing data' by deleting the attribute.

Examples

```
>>> f.shape
(12, 73, 96)
>>> m = f.mask
>>> type(m)
<cf.data.Data>
>>> m.dtype
dtype('bool')
>>> m.shape
[12, 73, 96]
>>> m.array.shape
(12, 73, 96)

>>> del f.mask
>>> f.array.mask
False
>>> import numpy
>>> f.array.mask is numpy.ma.nomask
True
```

measure

The name of spatial measure being represented.

Examples

```
>>> c.measure
'area'
```

missing_value

The missing_value CF attribute.

This attribute is forced to be consistent with the *_FillValue* attribute as follows:

- Assigning a value to the *missing_value* attribute also assigns the same value to the *_FillValue* attribute whether the latter has been previously set or not.
- Assigning a value to the *_FillValue* attribute also assigns the same value to the *missing_value* attribute, but only if the latter has previously been defined.

Examples

```
>>> f.missing_value = 1e30
>>> f.missing_value
1e30
>>> f._FillValue
1e30
>>> del f.missing_value
>>> f._FillValue
1e30
```

month_lengths

The month_lengths CF attribute.

ndim

Number of dimensions in the data array.

Examples

```
>>> f.shape
(73, 96)
>>> f.ndim
2
```

scale_factor

The scale_factor CF attribute.

This attribute is only used when writing to a file on disk.

Examples

```
>>> f.scale_factor = 10.0
>>> f.scale_factor
10.0
>>> del f.scale_factor
```

shape

Tuple of the data array's dimension sizes.

Examples

```
>>> f.shape
(73, 96)
```

size

Number of elements in the data array.

Examples

```
>>> f.shape
(73, 96)
>>> f.size
7008
```

standard_name

The standard_name CF attribute.

Examples

```
>>> f.standard_name = 'time'
>>> f.standard_name
'time'
>>> del f.standard_name
```

subset

Return a new variable whose data is subsetted.

This attribute may be indexed to select a subset from dimension index values.

Subsetting by indexing

Subsetting by dimension indices uses an extended Python slicing syntax, which is similar numpy array indexing. There are two extensions to the numpy indexing functionality:

- Size 1 dimensions are never removed.

  An integer index i takes the i-th element but does not reduce the rank of the output array by one.

- When advanced indexing is used on more than one dimension, the advanced indices work independently.

  When more than one dimension's slice is a 1-d boolean array or 1-d sequence of integers, then these indices work independently along each dimension (similar to the way vector subscripts work in Fortran), rather than by their elements.

Examples

units

The units CF attribute.

This attribute is a mirror of the units stored in the *Units* attribute.

Examples

```
>>> f.units = 'K'
>>> f.units
'K'
>>> del f.units
```

valid_max
The valid_max CF attribute.

Examples

```
>>> f.valid_max = 100.0
>>> f.valid_max
100.0
>>> del f.valid_max
```

valid_min
The valid_min CF attribute.

Examples

```
>>> f.valid_min = 100.0
>>> f.valid_min
100.0
>>> del f.valid_min
```

valid_range
The valid_range CF attribute.

May be set as a numpy array, a list or a tuple. Always returned as a tuple.

Examples

```
>>> f.valid_range = array([100., 400.])
>>> f.valid_range
(100.0, 400.0)
>>> del f.valid_range
>>> f.valid_range = (50., 450.)
```

varray
A numpy array view of the data array.

Changing the elements of the returned view changes the data array.

Examples

```
>>> a = f.varray
>>> type(a)
<type 'numpy.ndarray'>
>>> a
array([0, 1, 2, 3, 4])
>>> a[0] = 999
>>> f.varray
array([999, 1, 2, 3, 4])
```

8.2.2 cf.CellMethods

class cf.**CellMethods**(*string=None*)
Bases: cf.utils.CfList

A CF cell methods object to describe the characteristic of a field that is represented by cell values.

Each cell method is stored in a dictionary and these dictionaries are stored in a list-like object. Similarly to a CF cell_methods string, the order of cell methods in the list is important.

The dictionary representing each cell method recognizes the following keys (where all keys are optional; the referenced sections are from the NetCDF Climate and Forecast (CF) Metadata Conventions and words in double quotes ("...") refer to CF cell_methods components in the referenced sections):

Key	Value
method	A "method" component. Refer to section 7.3.
name	A list of all of the "name" components involved with the "method" component. Each element is either: •The standard_name of the cell method's dimension •The string 'area' •A netCDF variable name or netCDF dimension name The last option only occurs if the cell method's dimension either has no dimension coordinate or has a dimension coordinate with no standard_name. This is a deviation from the "name" components as described in sections 7.3, 7.3.1 and 7.3.4, in that standard_names are stored whenever possible, rather than only in the special cases described in section 7.3.4. Inspection of the 'no_coords' list retains the information required to fully interpret the cell method.
dim	A list of space dimension names corresponding to the 'name' list. A 'name' list value of 'area' always corresponds to a 'dim' list value of None.
no_coords	A list of booleans, corresponding to the 'name' list, indicating whether or not a particular cell method is relevant to the data in a way which may not be precisely defined by the corresponding dimension or dimensions. Refer to section 7.3.4.
interval	A list of Data objects giving "interval" component values, with their associated units, corresponding to the 'name' list. Refer to section 7.3.2.
comment	A "comment" or non-standardised information, component. Refer to sections 7.3.2 and 7.4
within	A "within" climatology component. Refer to section 7.4.
over	An "over" cell portion or "over" climatology component. Refer to sections 7.3.3 and 7.4.
where	A "where" component. Refer to section 7.3.3.

Note that the above table assumes that the cell methods have been constructed in the context of a field, the only way in which the 'dim' and 'no_coords' keys may take sensible values. If this is not the case, then elements of the 'dim' key default to False and elements of the 'no_coords' key default to None.

Examples

```
>>> c = cf.CellMethods('time: minimum within years time: mean over years (ENSO years)')
>>> print c
Cell methods    : time: minimum within years
                  time: mean over years (ENSO years)
>>> list(c)
[
 {'method'   : 'minimum',
  'name'     : ['time'],
  'dim'      : [False],
  'no_coords': [None],
  'within'   : 'years'
 },
 {'method'   : 'mean'
  'name'     : ['time'],
  'dim'      : [False],
```

```
  'no_coords': [None],
  'comment'  : 'ENSO years',
  'over'     : 'years'
 }
]

>>> c = c.CellMethods('lat: lon: standard_deviation')
>>> list(c)
[
 {'dim'      : [False, False],
  'method'   : 'standard_deviation'
  'name'     : ['lat', 'lon'],
  'no_coords': [None, None],
 }
]
```

Initialization

Parameters

string [str, optional] Initialize new instance from a CF-netCDF-like cell methods string. See the *parse* method for details. By default an empty cell methods is created.

Examples

```
>>> c = CellMethods()
>>> c = CellMethods('time: mean')
```

copy()

Return a deep copy.

Equivalent to `copy.deepcopy(s)`.

Returns

out [] The deep copy.

Examples

```
>>> s.copy()
```

count(*value*)

Return the number of occurrences of a given value.

Parameters

value : The value to count.

Returns

out [int] The number of occurrences of *value*.

Examples

```
>>> s
[1, 2, 3, 2, 4, 2]
>>> s.count(1)
1
>>> s.count(2)
3
```

dump(*id=None*)

Return a string containing a full description of the instance.

If a cell methods 'name' is followed by a '*' then that cell method is relevant to the data in a way which may not be precisely defined its corresponding dimension or dimensions.

Parameters

id [str, optional] Set the common prefix of component names. By default the instance's class name is used.

Returns

out [str] A string containing the description.

Examples

```
>>> x = c.dump()
>>> print c.dump()
>>> print c.dump(id='cellmethods1')
```

equals(*other*, *rtol=None*, *atol=None*, *traceback=False*)

True if two cell methods are logically equal, False otherwise.

Keys 'dim' and 'no_coords' are ignored in the comparison.

Parameters

other [] The object to compare for equality.

atol [float, optional] The absolute tolerance for all numerical comparisons, By default the value returned by the *ATOL* function is used.

rtol [float, optional] The relative tolerance for all numerical comparisons, By default the value returned by the *RTOL* function is used.

traceback [bool, optional] If True then print a traceback highlighting where the two instances differ.

Returns

out [bool] Whether or not the two instances are equal.

Examples

has_cellmethods(*other*)

Return True if and only if this cell methods is a super set of another.

Parameters

other [CellMethods] The other cell methods for comparison.

Returns

out [bool] Whether or not this cell methods is a super set of the other.

Examples

```
>>> c = cf.CellMethods('time: mean height: mean area: mean')
>>> d = cf.CellMethods('area: mean time: mean')
>>> c.has_cellmethods(d)
True
```

index(*value*, *start=0*, *stop=None*)

Return the first index of a given value.

Parameters value :

start : int, optional

stop : int, optional

Returns out : int

Raises

ValueError : If the given value is not in the list.

Examples

```
>>> s
[1, 2, 3, 2, 4, 2]
>>> s.index(1)
1
>>> s.index(2, start=2)
3
>>> s.index(2, start=2, stop=4)
3
```

insert(*index*, *object*)

Insert an object before the given index in place.

Parameters index : int

object :

Returns None

Examples

```
>>> s
[1, 2, 3]
>>> s.insert(1, 'A')
>>> s
[1, 'A', 2, 3]
```

netCDF_translation(*field*)

Translate netCDF variable names.

Parameters field : Field

The field which provides the translation.

Returns

out [CellMethods] A new cell methods instance with translated names.

Examples

```
>>> c = CellMethods('time: mean lon: mean')
>>> d = c.netCDF_translation(f)
```

parse(*string=None*)

Parse a CF cell_methods string into this CellMethods instance in place.

Parameters

string [str, optional] The CF cell_methods string to be parsed into the CellMethods object. By default the cell methods will be empty.

Returns None

Examples

```
>>> c = cf.CellMethods()
>>> c = c.parse('time: minimum within years time: mean over years (ENSO years)')
>>> print c
Cell methods    : time: minimum within years
                  time: mean over years (ENSO years)
```

strings()

Return a list of a CF-netCDF-like string of each cell method.

Note that if the intention is to concatenate the output list into a string for creating a netCDF(CF) cell_methods attribute, then the cell methods "name" components may need to be modified, where appropriate, to reflect netCDF variable names.

Returns

out [list] A string for each cell method.

Examples

```
>>> c = cf.CellMethods('time: minimum within years time: mean over years (ENSO years)')
>>> c.strings()
['time: minimum within years',
 'time: mean over years (ENSO years)']
```

8.2.3 cf.Coordinate

class `cf.`**`Coordinate`**(*attributes={}*)

Bases: `cf.variable.Variable`

A CF dimension or auxiliary coordinate construct.

The coordinate array's bounds (if present) are stored in the *bounds* attribute. The *climatology* attribute indicates if the bounds are intervals of climatological time.

If the coordinate is connected to a transform construct then a pointer to the transformation is stored in the *transform* attribute.

Initialization

Parameters

attributes [dict, optional] Initialize a new instance with attributes from the dictionary's key/value pairs. Values are deep copied.

`binary_mask`()

return new variable containing binary mask

`chunk`(*chunksize=None*, *extra_boundaries=None*, *adim=None*)

`copy`()

Return a deep copy.

Equivalent to `copy.deepcopy(c)`

Returns

out [] The deep copy.

Examples

```
>>> d = c.copy()
```

`delattr`(*attr*)

Delete a standard or non-standard CF attribute.

Parameters

attr [str] The name of the attribute to delete.

Returns None

Raises

AttributeError : If the variable does not have the named attribute.

Examples

```
>>> f.delattr('standard_name')
>>> f.delattr('foo')
```

`dump`(*id=None*, *omit=()*)

Return a string containing a full description of the coordinate.

Parameters

id: str, optional Set the common prefix of variable component names. By default the instance's class name is used.

omit [sequence of strs] Omit the given attributes from the description.

Returns

out [str] A string containing the description.

Examples

```
>>> x = c.dump()
>>> print c.dump()
>>> print c.dump(id='time_coord')
>>> print c.dump(omit=('long_name',))
```

equals(*other*, *rtol=None*, *atol=None*, *traceback=False*, *ignore=set([])*)
True if two variables are logically equal, False otherwise.

Parameters

other [] The object to compare for equality.

atol [float, optional] The absolute tolerance for all numerical comparisons, By default the value returned by the *ATOL* function is used.

rtol [float, optional] The relative tolerance for all numerical comparisons, By default the value returned by the *RTOL* function is used.

traceback [bool, optional] If True then print a traceback highlighting where the two instances differ.

ignore [set, optional] Omit these attributes from the comparison.

Returns

out [bool] Whether or not the two instances are equal.

Examples

```
>>> f.equals(f)
True
>>> g = f + 1
>>> f.equals(g)
False
>>> g -= 1
>>> f.equals(g)
True
>>> f.setattr('name', 'name0')
>>> g.setattr('name', 'name1')
>>> f.equals(g)
False
```

expand_dims(*axis*, *dim*, *direction*)
axis is an integer dim is a string direction is a boolean

extract(**args*, ***kwargs*)
Return the instance if it matches the given conditions.

Equivalent to:

```
def extract(f, *args, **kwargs):
    if f.match(*args, **kwargs):
        return f
    raise ValueError('')
```

Parameters

args, kwargs : As for the *match* method.

Returns

out : The variable as an object identity, if it matches the given conditions.

Raises

ValueError : If the variable does not match the conditions.

getattr(*attr*, **default*)

Get a standard or non-standard CF attribute.

Parameters

attr [str] The name of the attribute to get.

default [optional] Return *default* if and only if the variable does not have the named attribute.

Returns

out : The value of the named attribute, or the default value.

Raises

AttributeError : If the variable does not have the named attribute a default value has not been set.

Examples

```
>>> f.getattr('standard_name')
>>> f.getattr('standard_name', None)
>>> f.getattr('foo')
```

hasattr(*attr*)

Return True if the variable has standard or non-standard CF attribute.

Parameters

attr [str] The name of the attribute.

Returns

out [bool] True if the instance has the attribute.

Examples

```
>>> f.hasattr('standard_name')
>>> f.hasattr('foo')
```

match(*attr={}*, *priv={}*)

Determine whether or not a variable matches conditions on its attributes.

Parameters

attr [dict] Dictionary for which each key/value pair is a standard or non-standard CF attribute name and a condition for the attribute to be tested against. If the value is a sequence of conditions then the attribute matches if at least one of the conditions is passed.

In general, a condition is any object and it is passed if the attribute is equal to the object, with the following exceptions:

- If the attribute is string-valued, then the condition may be a regular expression pattern recognised by the *re* module and the condition is passed if the attribute matches the regular expression. Special characters for the start and end of the string are assumed and need not be included. For example, '.*wind' is equivalent to '^.*wind$'.
- For the 'Units' attribute, the condition is passed if the attribute is equivalent (rather than equal) to the object. (Note that this behaviour does not apply to the 'units' attribute.)

priv [dict, optional] Dictionary for which each key/value pair is an attribute name other than their standard and non-standard CF attribute and a condition for the attribute to be tested against. If the value is a sequence of conditions then the attribute matches if at least one of the conditions is passed.

In general, a condition behaves as for *attr*, with the following exception:

- For the *Units* attribute, the condition is passed if the attribute is equivalent (rather than equal) to the object. (Note that this behaviour does not apply to the *units* attribute.)

Returns

out [bool] Whether or not the variable matches the given criteria.

Examples

name(*long_name=False*, *ncvar=False*, *default=None*)

Return a name for the variable.

Returns the standard_name, long_name (if requested) or netCDF variable name (if requested), whichever it finds first, otherwise returns a default name.

Parameters long_name : bool, optional

If True, return the long_name if standard_name does not exist.

ncvar : bool, optional

If True, return ncvar if neither the standard_name nor long_name has already been returned.

default : str, optional

Return default if neither standard_name, long_name nor ncvar has already been returned.

Returns

name [str] The name of the variable.

override_units(*new_units*)

Override the data array units in place.

Not to be confused with setting the *Units* attribute to units which are equivalent to the original units.

This is different to setting the *Units* attribute, as the new units need not be equivalent to the original ones and the data array elements will not be changed to reflect the new units.

Parameters

new_units [str or Units] The new units for the data array.

Returns None

Examples

```
>>> f.Units
<CF Units: hPa>
>>> f.first_datum
100000.0
>>> f.override_units('km')
>>> f.Units
<CF Units: km>
>>> f.first_datum
100000.0
>>> f.override_units(cf.Units('watts'))
>>> f.Units
<CF Units: watts>
```

```
>>> f.first_datum
100000.0
```

reverse_dims(*axes=None*)

setattr(*attr*, *value*)
Set a standard or non-standard CF attribute.

Parameters

attr [str] The name of the attribute to set.

value : The value for the attribute.

Returns None

Examples

```
>>> f.setattr('standard_name', 'time')
>>> f.setattr('foo', 12.5)
```

squeeze(*axes=None*)
Remove size 1 dimensions from the coordinate's data array and bounds in place.

Parameters

axes [int or sequence of ints, optional] The size 1 axes to remove. By default, all size 1 axes are removed. Size 1 axes for removal may be identified by the integer positions of dimensions in the data array.

Returns None

Examples

```
>>> c.squeeze()
>>> c.squeeze(1)
>>> c.squeeze([1, 2])
```

transpose(*axes=None*)
Permute the dimensions of the coordinate's data array and bounds in place.

Parameters

axes [sequence of ints, optional] The new order of the data array. By default, reverse the dimensions' order, otherwise the axes are permuted according to the values given. The values of the sequence comprise the integer positions of the dimensions in the data array in the desired order.

Returns None

Examples

```
>>> c.transpose()
>>> c.ndim
3
>>> c.transpose([1, 2, 0])
```

Data
The *Data* object containing the data array.

Examples

```
>>> f.Data
<CF Data: >
```

Units
The Units object containing the units of the data array.

add_offset

The add_offset CF attribute.

This attribute is only used when writing to a file on disk.

Examples

```
>>> f.add_offset = -4.0
>>> f.add_offset
-4.0
>>> del f.add_offset
```

array

A numpy array deep copy of the data array.

Changing the returned numpy array does not change the data array.

Examples

```
>>> a = f.array
>>> type(a)
<type 'numpy.ndarray'>
>>> a
array([0, 1, 2, 3, 4])
>>> a[0] = 999
>>> f.array
array([0, 1, 2, 3, 4])
```

attributes

A dictionary of the standard and non-standard CF attributes.

Note that modifying the returned dictionary will not change the attributes.

Examples

```
>>> f.attributes
{'Conventions': 'CF-1.0',
 '_FillValue': 1e+20,
 'cell_methods': <CF CellMethods: time: mean>,
 'experiment_id': 'stabilization experiment (SRES A1B)',
 'long_name': 'Surface Air Temperature',
 'standard_name': 'AIR_TEMP',
 'title': 'SRES A1B',
 'units': 'K'}
```

axis

The axis CF attribute.

Examples

```
>>> c.axis = 'Y'
>>> c.axis
'Y'
>>> del c.axis
```

bounds

The *CoordinateBounds* object containing the coordinate array's cell bounds.

Examples

```
>>> c.bounds
<CF CoordinateBounds: >
```

calendar

The calendar CF attribute.

This attribute is a mirror of the calendar stored in the *Units* attribute.

Examples

```
>>> c.calendar = 'noleap'
>>> c.calendar
'noleap'
>>> del c.calendar
```

climatology

Indicator for the coordinate array's bounds representing climatological time intervals.

If not set then bounds are assumed to not represent climatological time intervals.

Examples

```
>>> c.climatology = True
>>> c.climatology
True
>>> del c.climatology
```

comment

The comment CF attribute.

Examples

```
>>> f.comment = 'a comment'
>>> f.comment
'a comment'
>>> del f.comment
```

dtype

Numpy data-type of the data array.

Examples

```
>>> f.dtype
dtype('float64')
```

first_datum

The first element of the data array.

Equivalent to `f.subset[(0,)*f.ndim].array.item()`

Examples

```
>>> f.array
array([[1, 2],
       [3, 4]])
>>> f.first_datum
1
```

history

The history CF attribute.

Examples

```
>>> f.history = 'created on 2012/10/01'
>>> f.history
'created on 2012/10/01'
>>> del f.history
```

is_scalar

True if and only if the data array is a scalar array.

Examples

```
>>> f.array
array(2)
```

```
>>> f.is_scalar
True

>>> f.array
array([2])
>>> f.is_scalar
False
```

last_datum
The last element of the data array.

Equivalent to `f.subset[(-1,)*f.ndim].array.item()`

Examples

```
>>> f.array
array([[1, 2],
       [3, 4]])
>>> f.first_datum
4
```

leap_month
The leap_month CF attribute.

Examples

```
>>> f.leap_month = 2
>>> f.leap_month
2
>>> del f.leap_month
```

leap_year
The leap_year CF attribute.

Examples

```
>>> f.leap_year = 1984
>>> f.leap_year
1984
>>> del f.leap_year
```

long_name
The long_name CF attribute.

Examples

```
>>> f.long_name = 'zonal_wind'
>>> f.long_name
'zonal_wind'
>>> del f.long_name
```

mask
The boolean missing data mask of the data array.

Returned as a *Data* object. The mask may be set to 'no missing data' by deleting the attribute.

Examples

```
>>> f.shape
(12, 73, 96)
>>> m = f.mask
>>> type(m)
<cf.data.Data>
>>> m.dtype
dtype('bool')
>>> m.shape
[12, 73, 96]
```

```
>>> m.array.shape
(12, 73, 96)

>>> del f.mask
>>> f.array.mask
False
>>> import numpy
>>> f.array.mask is numpy.ma.nomask
True
```

missing_value

The missing_value CF attribute.

This attribute is forced to be consistent with the *_FillValue* attribute as follows:

- Assigning a value to the *missing_value* attribute also assigns the same value to the *_FillValue* attribute whether the latter has been previously set or not.
- Assigning a value to the *_FillValue* attribute also assigns the same value to the *missing_value* attribute, but only if the latter has previously been defined.

Examples

```
>>> f.missing_value = 1e30
>>> f.missing_value
1e30
>>> f._FillValue
1e30
>>> del f.missing_value
>>> f._FillValue
1e30
```

month_lengths

The month_lengths CF attribute.

ndim

Number of dimensions in the data array.

Examples

```
>>> f.shape
(73, 96)
>>> f.ndim
2
```

positive

The positive CF attribute.

Examples

```
>>> c.positive = 'up'
>>> c.positive
'up'
>>> del c.positive
```

scale_factor

The scale_factor CF attribute.

This attribute is only used when writing to a file on disk.

Examples

```
>>> f.scale_factor = 10.0
>>> f.scale_factor
10.0
>>> del f.scale_factor
```

shape
Tuple of the data array's dimension sizes.

Examples

```
>>> f.shape
(73, 96)
```

size
Number of elements in the data array.

Examples

```
>>> f.shape
(73, 96)
>>> f.size
7008
```

standard_name
The standard_name CF attribute.

Examples

```
>>> f.standard_name = 'time'
>>> f.standard_name
'time'
>>> del f.standard_name
```

subset
Return a new coordinate whose data and bounds are subsetted in a consistent manner.

This attribute may be indexed to select a subset from dimension index values.

Subsetting by indexing

Subsetting by dimension indices uses an extended Python slicing syntax, which is similar numpy array indexing. There are two extensions to the numpy indexing functionality:

- Size 1 dimensions are never removed.

  An integer index i takes the i-th element but does not reduce the rank of the output array by one.

- When advanced indexing is used on more than one dimension, the advanced indices work independently.

  When more than one dimension's slice is a 1-d boolean array or 1-d sequence of integers, then these indices work independently along each dimension (similar to the way vector subscripts work in Fortran), rather than by their elements.

Examples

transform
Pointer to coordinate's transform.

units
The units CF attribute.

This attribute is a mirror of the units stored in the *Units* attribute.

Examples

```
>>> c.units = 'degrees_east'
>>> c.units
'degree_east'
>>> del c.units
```

valid_max
The valid_max CF attribute.

Examples

```
>>> f.valid_max = 100.0
>>> f.valid_max
100.0
>>> del f.valid_max
```

valid_min
The valid_min CF attribute.

Examples

```
>>> f.valid_min = 100.0
>>> f.valid_min
100.0
>>> del f.valid_min
```

valid_range
The valid_range CF attribute.

May be set as a numpy array, a list or a tuple. Always returned as a tuple.

Examples

```
>>> f.valid_range = array([100., 400.])
>>> f.valid_range
(100.0, 400.0)
>>> del f.valid_range
>>> f.valid_range = (50., 450.)
```

varray
Return a numpy view of the data.

Making changes to elements of the returned view changes the underlying data. Refer to `numpy.ndarray.view`[1].

Examples

```
>>> a = c.varray
>>> type(a)
<type 'numpy.ndarray'>
>>> a
array([0, 1, 2, 3, 4])
>>> a[0] = 999
>>> c.varray
array([999, 1, 2, 3, 4])
```

8.2.4 cf.CoordinateBounds

class `cf.`**CoordinateBounds**(*attributes={}*)
Bases: `cf.variable.Variable`

A CF coordinate's bounds object containing cell boundaries or intervals of climatological time. The containing coordinate's *climatology* attribute indicates which type of bounds are present.

Initialization

Parameters

attributes [dict, optional] Initialize a new instance with attributes from the dictionary's key/value pairs. Values are deep copied.

binary_mask()
return new variable containing binary mask

[1] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.view.html#numpy.ndarray.view

chunk(*chunksize=None*, *extra_boundaries=[]*, *adim=[]*)
Partition the data array using Large Amounts of Massive Arrays (LAMA) functionality.

Parameters chunksize : int

extra_boundaries : list

adim : list

Returns extra_boundaries, adim : {list, list}

copy(*_omit_special=()*)
Return a deep copy.

Equivalent to `copy.deepcopy(f)`.

Returns

out : The deep copy.

Examples

```
>>> g = f.copy()
```

delattr(*attr*)
Delete a standard or non-standard CF attribute.

Parameters

attr [str] The name of the attribute to delete.

Returns None

Raises

AttributeError : If the variable does not have the named attribute.

Examples

```
>>> f.delattr('standard_name')
>>> f.delattr('foo')
```

dump(*id=None*, *omit=()*)
Return a string containing a full description of the instance.

Parameters

id [str, optional] Set the common prefix of component names. By default the instance's class name is used.

omit [sequence of strs] Omit the given attributes from the description.

Returns

out [str] A string containing the description.

Examples

```
>>> x = v.dump()
>>> print v.dump()
>>> print v.dump(id='variable1')
>>> print v.dump(omit=('long_name',))
```

equals(*other*, *rtol=None*, *atol=None*, *traceback=False*, *ignore=set([])*)
True if two variables are logically equal, False otherwise.

Parameters

other [] The object to compare for equality.

atol [float, optional] The absolute tolerance for all numerical comparisons, By default the value returned by the *ATOL* function is used.

rtol [float, optional] The relative tolerance for all numerical comparisons, By default the value returned by the *RTOL* function is used.

traceback [bool, optional] If True then print a traceback highlighting where the two instances differ.

ignore [set, optional] Omit these attributes from the comparison.

Returns

out [bool] Whether or not the two instances are equal.

Examples

```
>>> f.equals(f)
True
>>> g = f + 1
>>> f.equals(g)
False
>>> g -= 1
>>> f.equals(g)
True
>>> f.setattr('name', 'name0')
>>> g.setattr('name', 'name1')
>>> f.equals(g)
False
```

expand_dims(*axis*, *dim*, *direction*)
axis is an integer dim is a string direction is a boolean

extract(**args*, ***kwargs*)
Return the instance if it matches the given conditions.

Equivalent to:

```
def extract(f, *args, **kwargs):
    if f.match(*args, **kwargs):
        return f
    raise ValueError('')
```

Parameters

args, kwargs : As for the *match* method.

Returns

out : The variable as an object identity, if it matches the given conditions.

Raises

ValueError : If the variable does not match the conditions.

getattr(*attr*, **default*)
Get a standard or non-standard CF attribute.

Parameters

attr [str] The name of the attribute to get.

default [optional] Return *default* if and only if the variable does not have the named attribute.

Returns

out : The value of the named attribute, or the default value.

Raises

AttributeError : If the variable does not have the named attribute a default value has not been set.

Examples

```
>>> f.getattr('standard_name')
>>> f.getattr('standard_name', None)
>>> f.getattr('foo')
```

hasattr(*attr*)

Return True if the variable has standard or non-standard CF attribute.

Parameters

attr [str] The name of the attribute.

Returns

out [bool] True if the instance has the attribute.

Examples

```
>>> f.hasattr('standard_name')
>>> f.hasattr('foo')
```

match(*attr={}*, *priv={}*)

Determine whether or not a variable matches conditions on its attributes.

Parameters

attr [dict] Dictionary for which each key/value pair is a standard or non-standard CF attribute name and a condition for the attribute to be tested against. If the value is a sequence of conditions then the attribute matches if at least one of the conditions is passed.

In general, a condition is any object and it is passed if the attribute is equal to the object, with the following exceptions:

- If the attribute is string-valued, then the condition may be a regular expression pattern recognised by the *re* module and the condition is passed if the attribute matches the regular expression. Special characters for the start and end of the string are assumed and need not be included. For example, '.*wind' is equivalent to '^.*wind$'.
- For the 'Units' attribute, the condition is passed if the attribute is equivalent (rather than equal) to the object. (Note that this behaviour does not apply to the 'units' attribute.)

priv [dict, optional] Dictionary for which each key/value pair is an attribute name other than their standard and non-standard CF attribute and a condition for the attribute to be tested against. If the value is a sequence of conditions then the attribute matches if at least one of the conditions is passed.

In general, a condition behaves as for *attr*, with the following exception:

- For the *Units* attribute, the condition is passed if the attribute is equivalent (rather than equal) to the object. (Note that this behaviour does not apply to the *units* attribute.)

Returns

out [bool] Whether or not the variable matches the given criteria.

Examples

name(*long_name=False*, *ncvar=False*, *default=None*)

Return a name for the variable.

Returns the standard_name, long_name (if requested) or netCDF variable name (if requested), whichever it finds first, otherwise returns a default name.

Parameters long_name : bool, optional

If True, return the long_name if standard_name does not exist.

ncvar : bool, optional

If True, return ncvar if neither the standard_name nor long_name has already been returned.

default : str, optional

Return default if neither standard_name, long_name nor ncvar has already been returned.

Returns

name [str] The name of the variable.

reverse_dims(*axes=None*)

Reverse the directions of the data array dimensions in place.

Equivalent to indexing the specified dimensions with `::-1`.

Parameters

axes [int or sequence of ints] By default all dimensions are reversed.

Returns

out [list of ints] The axes which were reversed, in arbitrary order.

Examples

```
>>> f.reverse_dims()
>>> f.reverse_dims(1)
>>> f.reverse_dims([0, 1])
```

setattr(*attr*, *value*)

Set a standard or non-standard CF attribute.

Parameters

attr [str] The name of the attribute to set.

value : The value for the attribute.

Returns None

Examples

```
>>> f.setattr('standard_name', 'time')
>>> f.setattr('foo', 12.5)
```

squeeze(*axes=None*)

axes are integers

transpose(*axes=None*)

axes are integers

Data

The *Data* object containing the data array.

Examples

```
>>> f.Data
<CF Data: >
```

Units

The *Units* object containing the units of the data array.

Stores the units and calendar CF attributes in an internally consistent manner. These attributes are mirrored by the *units* and *calendar* attributes respectively.

Examples

```
>>> f.Units
<CF Units: K>
```

add_offset
The add_offset CF attribute.

This attribute is only used when writing to a file on disk.

Examples

```
>>> f.add_offset = -4.0
>>> f.add_offset
-4.0
>>> del f.add_offset
```

array
A numpy array deep copy of the data array.

Changing the returned numpy array does not change the data array.

Examples

```
>>> a = f.array
>>> type(a)
<type 'numpy.ndarray'>
>>> a
array([0, 1, 2, 3, 4])
>>> a[0] = 999
>>> f.array
array([0, 1, 2, 3, 4])
```

attributes
A dictionary of the standard and non-standard CF attributes.

Note that modifying the returned dictionary will not change the attributes.

Examples

```
>>> f.attributes
{'Conventions': 'CF-1.0',
 '_FillValue': 1e+20,
 'cell_methods': <CF CellMethods: time: mean>,
 'experiment_id': 'stabilization experiment (SRES A1B)',
 'long_name': 'Surface Air Temperature',
 'standard_name': 'AIR_TEMP',
 'title': 'SRES A1B',
 'units': 'K'}
```

calendar
The calendar CF attribute.

This attribute is a mirror of the calendar stored in the *Units* attribute.

Examples

```
>>> f.calendar = 'noleap'
>>> f.calendar
'noleap'
>>> del f.calendar
```

comment
The comment CF attribute.

Examples

```
>>> f.comment = 'a comment'
>>> f.comment
```

```
'a comment'
>>> del f.comment
```

dtype
Numpy data-type of the data array.

Examples

```
>>> f.dtype
dtype('float64')
```

first_datum
The first element of the data array.

Equivalent to `f.subset[(0,)*f.ndim].array.item()`

Examples

```
>>> f.array
array([[1, 2],
       [3, 4]])
>>> f.first_datum
1
```

history
The history CF attribute.

Examples

```
>>> f.history = 'created on 2012/10/01'
>>> f.history
'created on 2012/10/01'
>>> del f.history
```

is_scalar
True if and only if the data array is a scalar array.

Examples

```
>>> f.array
array(2)
>>> f.is_scalar
True

>>> f.array
array([2])
>>> f.is_scalar
False
```

last_datum
The last element of the data array.

Equivalent to `f.subset[(-1,)*f.ndim].array.item()`

Examples

```
>>> f.array
array([[1, 2],
       [3, 4]])
>>> f.first_datum
4
```

leap_month
The leap_month CF attribute.

Examples

```
>>> f.leap_month = 2
>>> f.leap_month
2
>>> del f.leap_month
```

leap_year

The leap_year CF attribute.

Examples

```
>>> f.leap_year = 1984
>>> f.leap_year
1984
>>> del f.leap_year
```

long_name

The long_name CF attribute.

Examples

```
>>> f.long_name = 'zonal_wind'
>>> f.long_name
'zonal_wind'
>>> del f.long_name
```

mask

The boolean missing data mask of the data array.

Returned as a *Data* object. The mask may be set to 'no missing data' by deleting the attribute.

Examples

```
>>> f.shape
(12, 73, 96)
>>> m = f.mask
>>> type(m)
<cf.data.Data>
>>> m.dtype
dtype('bool')
>>> m.shape
[12, 73, 96]
>>> m.array.shape
(12, 73, 96)

>>> del f.mask
>>> f.array.mask
False
>>> import numpy
>>> f.array.mask is numpy.ma.nomask
True
```

missing_value

The missing_value CF attribute.

This attribute is forced to be consistent with the *_FillValue* attribute as follows:

- Assigning a value to the *missing_value* attribute also assigns the same value to the *_FillValue* attribute whether the latter has been previously set or not.
- Assigning a value to the *_FillValue* attribute also assigns the same value to the *missing_value* attribute, but only if the latter has previously been defined.

Examples

```
>>> f.missing_value = 1e30
>>> f.missing_value
```

```
1e30
>>> f._FillValue
1e30
>>> del f.missing_value
>>> f._FillValue
1e30
```

month_lengths
The month_lengths CF attribute.

ndim
Number of dimensions in the data array.

Examples

```
>>> f.shape
(73, 96)
>>> f.ndim
2
```

scale_factor
The scale_factor CF attribute.

This attribute is only used when writing to a file on disk.

Examples

```
>>> f.scale_factor = 10.0
>>> f.scale_factor
10.0
>>> del f.scale_factor
```

shape
Tuple of the data array's dimension sizes.

Examples

```
>>> f.shape
(73, 96)
```

size
Number of elements in the data array.

Examples

```
>>> f.shape
(73, 96)
>>> f.size
7008
```

standard_name
The standard_name CF attribute.

Examples

```
>>> f.standard_name = 'time'
>>> f.standard_name
'time'
>>> del f.standard_name
```

subset
Return a new variable whose data is subsetted.

This attribute may be indexed to select a subset from dimension index values.

Subsetting by indexing

Subsetting by dimension indices uses an extended Python slicing syntax, which is similar numpy array indexing. There are two extensions to the numpy indexing functionality:

•Size 1 dimensions are never removed.

An integer index i takes the i-th element but does not reduce the rank of the output array by one.

•When advanced indexing is used on more than one dimension, the advanced indices work independently.

When more than one dimension's slice is a 1-d boolean array or 1-d sequence of integers, then these indices work independently along each dimension (similar to the way vector subscripts work in Fortran), rather than by their elements.

Examples

units

The units CF attribute.

This attribute is a mirror of the units stored in the *Units* attribute.

Examples

```
>>> f.units = 'K'
>>> f.units
'K'
>>> del f.units
```

valid_max

The valid_max CF attribute.

Examples

```
>>> f.valid_max = 100.0
>>> f.valid_max
100.0
>>> del f.valid_max
```

valid_min

The valid_min CF attribute.

Examples

```
>>> f.valid_min = 100.0
>>> f.valid_min
100.0
>>> del f.valid_min
```

valid_range

The valid_range CF attribute.

May be set as a numpy array, a list or a tuple. Always returned as a tuple.

Examples

```
>>> f.valid_range = array([100., 400.])
>>> f.valid_range
(100.0, 400.0)
>>> del f.valid_range
>>> f.valid_range = (50., 450.)
```

varray

A numpy array view of the data array.

Changing the elements of the returned view changes the data array.

Examples

```
>>> a = f.varray
>>> type(a)
<type 'numpy.ndarray'>
>>> a
array([0, 1, 2, 3, 4])
>>> a[0] = 999
>>> f.varray
array([999, 1, 2, 3, 4])
```

8.2.5 cf.Data

class `cf.`**`Data`**(*data=None*, *units=None*, *_FillValue=None*, *chunk=True*)

Bases: `object`

An N-dimensional data array with units and masked values.

- Contains an N-dimensional, indexable and broadcastable array with many similarities to a numpy array.
- Contains the units of the array elements.
- Supports masked arrays, regardless of whether or not it was initialized with a masked array.
- Uses Large Amounts of Massive Arrays (LAMA) functionality to store and operate on arrays which are larger then the available memory.

Indexing

A data array is indexable in a similar way to numpy array indexing but for two important differences:

- Size 1 dimensions are never removed.

  An integer index i takes the i-th element but does not reduce the rank of the output array by one.

- When advanced indexing is used on more than one dimension, the advanced indices work independently.

  When more than one dimension's slice is a 1-d boolean array or 1-d sequence of integers, then these indices work independently along each dimension (similar to the way vector subscripts work in Fortran), rather than by their elements.

Examples

```
>>> d.shape
[12, 19, 73, 96]
>>> d[0, :, [0,1], [0,1,2]].shape
[1, 19, 2, 3]
```

Conversion to a numpy array

The data array may be converted to either a numpy array view or an independent numpy array of the underlying data with the *varray* and *array* attributes respectively. Changing a numpy array view in place will also change the data array. Note that the numpy array created with the *array* or *varray* attribute forces all of the data to be read into memory at the same time, which may not be possible for very large arrays.

Initialization

Parameters

data [array-like, optional] The data for the array.

units [str or Units, optional] The units of the data.

_FillValue [object, optional] The fill value of the data. By default the numpy fill value appropriate to the data type will be used.

chunk [bool, optional] If True then the data array will be partitioned if it is larger than the chunk size.

Examples

```
>>> d = cf.Data(5)
>>> d = cf.Data([1,2,3], units='K')
>>> import numpy
>>> d = cf.Data(numpy.arange(10).reshape(2,5), units=cf.Units('m/s'), _FillValue=-999)
>>> d = cf.Data(('f', 'l', 'y'))
```

add_partitions(*extra_boundaries*, *adim*, *existing_boundaries=None*)

Examples

```
>>> d.add_partitions(    )
```

all()

Test whether all array elements evaluate to True.

Masked values are considered as True during computation.

Examples

```
>>> d.array
array([0, 3, 0])
>>> d.all()
False

>>> d.array
array([1, 3, 2])
>>> d.all()
True
```

any()

Test whether any array elements evaluate to True.

Masked values are considered as True during computation.

Examples

```
>>> d.array
array([0, 0, 0])
>>> d.any()
False

>>> d.array
array([0, 3, 0])
>>> d.any()
True
```

change_dimension_names(*dim_name_map*)

Change the dimension names.

The dimension names are arbitrary (though unique), so mapping them to another arbitrary (though unique) set does not change the data array values, units, dimension directions nor dimension order.

Examples

```
>>> d.order
['dim0', 'dim1', 'dim2']
>>> dim_name_map
{'dim0': 'dim1',
 'dim1': 'dim0',
 'dim2': 'dim2',
 'dim3': 'dim3'}
>>> d.change_dimension_names(dim_name_map)
>>> d.order
['dim1', 'dim0', 'dim2']
```

chunk(*chunksize=None*, *extra_boundaries=None*, *chunk_dims=None*)

Parameters chunksize : int, optional

extra_boundaries : sequence of lists or tuples, optional

chunk_dims : sequence of lists or tuples, optional

Returns extra_boundaries, chunk_dims : {list, list}

Examples

```
>>> d.chunk()
>>> d.chunk(100000)
>>> d.chunk(extra_boundaries=([3, 6],), chunk_dims=['dim0'])
>>> d.chunk(extra_boundaries=([3, 6], [40, 80]), chunk_dims=['dim0', 'dim1'])
```

copy()
Return a deep copy.

Equivalent to `copy.deepcopy(d)`.

Returns

out [] The deep copy.

Examples

```
>>> e = d.copy()
```

dump(*id=None*, *omit=()*)

equals(*other*, *rtol=None*, *atol=None*, *traceback=False*)
True if two data arrays are logically equal, False otherwise.

Parameters

other [] The object to compare for equality.

atol [float, optional] The absolute tolerance for all numerical comparisons, By default the value returned by the *ATOL* function is used.

rtol [float, optional] The relative tolerance for all numerical comparisons, By default the value returned by the *RTOL* function is used.

traceback [bool, optional] If True then print a traceback highlighting where the two instances differ.

Returns

out [bool] Whether or not the two instances are equal.

Examples

```
>>> d.equals(d)
True
>>> d.equals(d + 1)
False
```

expand_aggregating_dims(*adim*)

expand_dims(*axis=0*, *dim='None'*, *direction=True*)
no check is done for dim already being in self.order

hash()
Return a hash value for the data array.

Note that generating the hash realizes the entire array in memory, which may not be possible for large arrays.

Returns

out [int] The hash value.

Examples

```
>>> d.hash()
3632586161869339209
```

new_dimension_name()

Return a dimension name not being used by the data array.

Note that a partition of the data array may have dimensions which don't belong to the data array itself.

Returns

out [str] The new dimension name.

Examples

```
>>> d.order
['dim1', 'dim0']
>>> d.partitions.info('order')
[['dim0', 'dim0'],
 ['dim1', 'dim0', 'dim2']]
>>> d.new_dimension_name()
'dim3'
```

override_units(*new_units*)

Override the data array units in place.

Not to be confused with setting the *Units* attribute to units which are equivalent to the original units.

This is different to setting the *Units* attribute, as the new units need not be equivalent to the original ones and the data array elements will not be changed to reflect the new units.

Parameters

new_units [str or Units] The new units for the data array.

Returns None

Examples

```
>>> d.Units
<CF Units: hPa>
>>> d.first_datum
100000.0
>>> d.override_units('km')
>>> d.Units
<CF Units: km>
>>> d.first_datum
100000.0
>>> d.override_units(cf.Units('watts'))
>>> d.Units
<CF Units: watts>
>>> d.first_datum
100000.0
```

partition_boundaries()

reverse_dims(*axes=None*)

Reverse the directions of data array dimensions in place.

Equivalent to indexing the specified dimensions with `::-1`.

Parameters

axes [int or sequence of ints] Reverse the dimensions whose positions are given. By default all dimensions are reversed.

Returns

out [list of ints] The axes which were reversed, in arbitrary order.

Examples

```
>>> d.ndim
3
>>> d.reverse_dims()
>>> d.reverse_dims(1)
>>> e = d[::-1, ::-1, :]
>>> d.reverse_dims([0, 1]).equals(e)
True
```

save_to_disk(*itemsize=None*)

Parameters itemsize : int, optional

Returns out : bool

Examples

```
>>>
>>>
```

set_location_map()

squeeze(*axes=None*)

Remove size 1 dimensions from the shape of the data in place.

Parameters axes : int or tuple of ints, optional

The axes to be squeezed given by their positions. If unset then all size one dimensions of the data array are removed.

Returns

out [tuple of ints] The axes which were squeezed as a tuple of their positions.

Examples

```
>>> v.shape
[1]
>>> v.squeeze()
>>> v.shape
[]

>>> v.shape
[1, 2, 1, 3, 1, 4, 1, 5, 1, 6, 1]
>>> v.squeeze(axis=2).shape
[1, 2, 3, 1, 4, 1, 5, 1, 6, 1]
>>> v.squeeze(axis=(0,)).shape
[2, 3, 1, 4, 1, 5, 1, 6, 1]
>>> v.squeeze(axis=(2, 4)).shape
[2, 3, 4, 5, 1, 6, 1]
>>> v.squeeze().shape
[2, 3, 4, 5, 6]
```

to_disk()

to_memory(*regardless=False*)

Store the data array in memory if it is smaller than the chunk size.

Parameters

regardless [bool, optional] If True then store the data array in memory regardless of its size.

Returns None

Examples

```
>>> d.to_memory()
>>> d.to_memory(True)
```

transpose(*axes=None*)

axes: list of ints, optional By default, reverse the dimensions, otherwise permute the axes according to the values given.

ufunc(*func*, **args*, ***kwargs*)
Return a

Units
Deleting the *Units* attribute actually sets it to undefined units, so the Data object is guaranteed to always have the *Units* attribute.

Examples

```
>>> del d.Units
>>> print d.Units
<CF Units: >
```

array
A numpy array copy the data array.

Examples

```
>>> a = d.array
>>> type(a)
<type 'numpy.ndarray'>
```

binary_mask
The binary missing data mask of the data array.

The binary mask has 0 where the data array has missing data and 1 otherwise.

Examples

```
>>> d.mask.array
array([ True, False,  True, False], dtype=bool)
>>> b = d.binary_mask.array
array([0, 1, 0, 1], dtype=int32)
```

dtype
Numpy data-type of the data array.

If the array is partitioned internally into sub-arrays with different data-types, then the normal data-type coercion rules apply. For example, if the partitions have data-types 'int32' and 'float32' then the data array's data-type will be 'float32'.

Examples

```
>>> type(f.dtype)
<type 'numpy.dtype'>
>>> f.dtype
dtype('float64')
```

first_datum
The first element of the data array.

May be retrieved or set.

Equivalent to `x[(0,) * x.ndim].array.item()` or `x[(0,) * x.ndim] = y`

Examples

```
>>> d.array
array([[1, 2],
       [3, 4]])
```

```
>> d.first_datum
1
>> d.first_datum = 999
>> d.array
array([[999,   2],
       [  3,   4]])
```

is_masked

True if the data array has any masked values.

Examples

```
>>> d.is_masked
True
```

is_scalar

True if the data array is a 0-d scalar array.

Examples

```
>>> d.ndim
0
>>> d.is_scalar
True

>>> d.ndim >= 1
True
>>> d.is_scalar
False
```

last_datum

The last element of the data array.

May be retrieved or set.

Equivalent to `x[(-1,) * x.ndim].array.item()` or `x[(-1,) * x.ndim] = y`

Examples

```
>>> d.array
array([[1, 2],
       [3, 4]])
>> d.last_datum
4
>> d.last_datum = 999
>> d.array
array([[  1,   2],
       [  3, 999]])
```

mask

The boolean missing data mask of the data array.

The boolean mask has True where the data array has missing data and False otherwise.

The mask may be set to the equivalent of 'no missing data' (i.e. all elements are False) by deleting the attribute.

Examples

```
>>> d.shape
[12, 73, 96]
>>> m = d.mask
>>> m.dtype
dtype('bool')
>>> m.shape
[12, 73, 96]
```

```
>>> del d.mask
>>> d.array.mask
False
>>> import numpy
>>> a.array.mask is numpy.ma.nomask
True
```

ndim
Number of dimensions in the data array.

Examples

```
>>> d.shape
[73, 96]
>>> d.ndim
2
```

shape
List of the data array's dimension sizes.

Note that this attribute is a list, not a tuple.

Examples

```
>>> d.shape
[73, 96]
```

size
Number of elements in the data array.

Examples

```
>>> d.shape
[73, 96]
>>> d.size
7008
```

varray
A numpy array view the data array.

Note that making changes to elements of the returned view changes the underlying data.

Examples

```
>>> a = d.varray
>>> type(a)
<type 'numpy.ndarray'>
>>> a
array([0, 1, 2, 3, 4])
>>> a[0] = 999
>>> d.varray
array([999, 1, 2, 3, 4])
```

8.2.6 cf.Flags

class `cf.`**`Flags`**(***kwargs*)
Bases: `object`

Self-describing CF flag values.

Stores the flag_values, flag_meanings and flag_masks CF attributes in an internally consistent manner.

Initialization

Parameters

flag_values [optional] The flag_values CF property. Sets the *flag_values* attribute.

flag_meanings [optional] The flag_meanings CF property. Sets the *flag_meanings* attribute.

flag_masks [optional] The flag_masks CF property. Sets the *flag_masks* attribute.

copy()

Return a deep copy.

Equivalent to `copy.deepcopy(f)`

Returns

out : The deep copy.

Examples

```
>>> f.copy()
```

dump(*id=None*)

Return a string containing a full description of the instance.

Parameters

id [str, optional] Set the common prefix of component names. By default the instance's class name is used.

Returns

out [str] A string containing the description.

Examples

```
>>> x = f.dump()
>>> print f.dump()
>>> print f.dump(id='flags1')
```

equals(*other*, *rtol=None*, *atol=None*, *traceback=False*)

True if two groups of flags are logically equal, False otherwise.

Note that both instances are sorted in place prior to the comparison.

Parameters

other [] The object to compare for equality.

atol [float, optional] The absolute tolerance for all numerical comparisons, By default the value returned by the *ATOL* function is used.

rtol [float, optional] The relative tolerance for all numerical comparisons, By default the value returned by the *RTOL* function is used.

traceback [bool, optional] If True then print a traceback highlighting where the two instances differ.

Returns

out [bool] Whether or not the two instances are equal.

Examples

```
>>> f
<CF Flags: flag_values=[1 0 2], flag_masks=[2 0 2], flag_meanings=['medium' 'low' 'high']
>>> g
<CF Flags: flag_values=[2 0 1], flag_masks=[2 0 2], flag_meanings=['high' 'low' 'medium']
>>> f.equals(g)
True
>>> f
<CF Flags: flag_values=[0 1 2], flag_masks=[0 2 2], flag_meanings=['low' 'medium' 'high']
```

```
>>> g
<CF Flags: flag_values=[0 1 2], flag_masks=[0 2 2], flag_meanings=['low' 'medium' 'high']
```

hash()
Return a hash value for the flags.

Note that the flags will be sorted in place.

Returns

out [int] The hash value.

Examples

```
>>> f.hash()
-956218661958673979
```

sort()
Sort the flags in place.

By default sort by flag values. If flag values are not present then sort by flag meanings. If flag meanings are not present then sort by flag_masks.

Returns None

Examples

```
>>> f
<CF Flags: flag_values=[2 0 1], flag_masks=[2 0 2], flag_meanings=['high' 'low' 'medium']
>>> f.sort()
>>> f
<CF Flags: flag_values=[0 1 2], flag_masks=[0 2 2], flag_meanings=['low' 'medium' 'high']
```

flag_masks
The flag_masks CF attribute.

Stored as a 1-d numpy array but may be set as array-like object.

Examples

```
>>> f.flag_masks = numpy.array([1, 2, 4], dtype='int8')
>>> f.flag_masks
array([1, 2, 4], dtype=int8)
>>> f.flag_masks = 1
>>> f.flag_masks
array([1])
```

flag_meanings
The flag_meanings CF attribute.

Stored as a 1-d numpy string array but may be set as a space delimited string or any array-like object.

Examples

```
>>> f.flag_meanings = 'low medium      high'
>>> f.flag_meanings
array(['low', 'medium', 'high'],
      dtype='|S6')
>>> f.flag_meanings = ['left', 'right']
>>> f.flag_meanings
array(['left', 'right'],
      dtype='|S5')
>>> f.flag_meanings = 'ok'
>>> f.flag_meanings
array(['ok'],
      dtype='|S2')
>>> f.flag_meanings = numpy.array(['a', 'b'])
>>> f.flag_meanings
```

```
array(['a', 'b'],
      dtype='|S1')
```

flag_values
The flag_values CF attribute.

Stored as a 1-d numpy array but may be set as any array-like object.

Examples

```
>>> f.flag_values = ['a', 'b', 'c']
>>> f.flag_values
array(['a', 'b', 'c'], dtype='|S1')
>>> f.flag_values = numpy.arange(4, dtype='int8')
>>> f.flag_values
array([1, 2, 3, 4], dtype=int8)
>>> f.flag_values = 1
>>> f.flag_values
array([1])
```

8.2.7 cf.Space

class cf.**Space**(**args*, ***kwargs*)
Bases: cf.utils.CfDict

Completely describe a field's coordinate system (space).

It contains the dimension constructs, auxiliary coordinate constructs, cell measure constructs and transform constructs defined by the CF data model.

The space is a dictionary-like object whose key/value pairs identify and store the coordinate and cell measure constructs which describe it.

The dimensionality of the space's components and its transforms are stored as attributes.

Initialization

The attributes *dimension_sizes*, *dimensions* and *transforms* are automatically initialized.

Parameters

args, kwargs Keys and values are initialized exactly as for a built-in dict. Keys are coordinate and cell measure construct identifiers (such as dim1, aux0, and cm2) and values are coordinate and cell measure instances.

aux_coords(*dim=None*)
Return a dictionary whose values are the auxiliary coordinates which span the given dimension and keys of the space's auxiliary coordinate identifiers.

Parameters

dim [str, optional] The identifier of the dimension to be spanned. By default all dimensions are considered (so all auxiliary coordinates are returned).

Returns

out [dict] The auxiliary coordinates and their identifiers.

Examples

```
>>> s.dimensions
{'data': ['dim0', 'dim1', 'dim2'],
 'dim0': ['dim0'],
 'dim1': ['dim1'],
 'dim2': ['dim2'],
 'aux0': ['dim1', 'dim2'],
 'aux1': ['dim0'],
```

```
 'aux2': ['dim2', 'dim1']}
>>> s.aux_coords()
{'aux0': <CF Coordinate: ...>,
 'aux1': <CF Coordinate: ...>,
 'aux2': <CF Coordinate: ...>}
>>> s.aux_coords('dim2')
{'aux0': <CF Coordinate: ...>,
 'aux2': <CF Coordinate: ...>}
```

cell_measures(*dim=None*)

Return a dictionary whose values are the cell measures which span the given dimension and keys of the space's cell measure identifiers.

Parameters

dim [str, optional] The identifier of the dimension to be spanned. By default all dimensions are considered (so all cell measures are returned).

Returns

out [dict] The cell measures and their identifiers.

Examples

```
>>> s.dimensions
{'data': ['dim0', 'dim1', 'dim2'],
 'dim0': ['dim0'],
 'dim1': ['dim1'],
 'dim2': ['dim2'],
 'cm0' : ['dim1', 'dim2'],
 'cm1' : ['dim1', 'dim2', 'dim3']}
>>> s.cell_measures()
{'cm0': <CF CellMeasures: ...>,
 'cm1': <CF CellMeasures: ...>}
>>> s.cell_measures('dim3')
{'cm1': <CF CellMeasures: ...>}
```

coord(*arg*, *role=None*, *key=False*, *exact=False*, *maximal_match=True*)

Find a coordinate of the space by name. Refer to `cf.Field.coord` for details.

copy()

Return a deep copy.

Equivalent to `copy.deepcopy(d)`.

Returns

out [] The deep copy.

Examples

```
>>> d.copy()
```

direction(*dim*)

Return True if a dimension is increasing, otherwise return False.

A dimension is considered to be increasing if its dimension coordinate values are increasing in index space or if it has no dimension coordinate.

The direction is taken directly from the appropriate coordinate's Data object, if available. (This is because we can assume that the space has been finalized.)

Parameters

dim [str] The identifier of the dimension (such as 'dim0').

Returns

out [bool] Whether or not the dimension is increasing.

Examples

```
>>> s.dimension_sizes
{'dim0': 3, 'dim1': 1, 'dim2': 2, 'dim3': 2, 'dim4': 99}
>>> s.dimensions
{'dim0': ['dim0'],
 'dim1': ['dim1'],
 'aux0': ['dim0'],
 'aux1': ['dim2'],
 'aux2': ['dim3'],
}
>>> s['dim0'].array
array([  0  30  60])
>>> s.direction('dim0')
True
>>> s['dim1'].array
array([15])
>>> s['dim1'].bounds.array
array([  30  0])
>>> s.direction('dim1')
False
>>> s['aux1'].array
array([0, -1])
>>> s.direction('dim2')
True
>>> s['aux2'].array
array(['z' 'a'])
>>> s.direction('dim3')
True
>>> s.direction('dim4')
True
```

dump(*id=None*)

Return a string containing a full description of the space.

Parameters

id [str, optional] Set the common prefix of component names. By default the instance's class name is used.

Returns

out [str] A string containing the description.

Examples

```
>>> x = s.dump()
>>> print s.dump()
>>> print s.dump(id='space1')
```

equals(*other*, *rtol=None*, *atol=None*, *traceback=False*)

True if two spaces are logically equal, False otherwise.

Equality is defined as follows:

- There is one-to-one correspondence between dimensions and dimension sizes between the two spaces.
- For each space component type (dimension coordinate, auxiliary coordinate and cell measures), the set of constructs in one space equals that of the other space. The component identifiers need not be the same.
- The set of transforms in one space equals that of the other space. The transform identifiers need not be the same.

Equality of numbers is to within a tolerance.

Parameters

other : The object to compare for equality.

atol [float, optional] The absolute tolerance for all numerical comparisons, By default the value returned by the *ATOL* function is used.

rtol [float, optional] The relative tolerance for all numerical comparisons, By default the value returned by the *RTOL* function is used.

traceback [bool, optional] If True then print a traceback highlighting where the two instances differ.

Returns

out [bool] Whether or not the two instances are equal.

Examples

```
>>> s.equals(t)
True
```

expand_dims(*coord=None*, *size=1*)

Expand the space with a new dimension in place.

The new dimension may by of any size greater then 0.

Parameters

coord [Coordinate, optional] A dimension coordinate for the new dimension. The new dimension's size is set to the size of the coordinate's array.

size [int, optional] The size of the new dimension. By default a dimension of size 1 is introduced. Ignored if *coord* is set.

Returns None

Examples

```
>>> s.expand_dims()
>>> s.expand_dims(size=12)
>>> c
<CF Coordinate: >
>>> s.expand_dims(coord=c)
```

get_keys(*regex=None*)

Return a list of the key names which match a regular expression.

Parameters

regex [str, optional] The regular expression with which to identify key names. By default all keys names are returned.

Returns

out [list] A list of key names.

Examples

```
>>> d.keys()
['dim2', 'dim0', 'dim1', 'aux0', 'cm0']
>>> d.get_keys()
['dim2', 'dim0', 'dim1', 'aux0', 'cm0']
>>> d.get_keys('dim')
['dim2', 'dim0', 'dim1']
>>> d.get_keys('^aux|^dim')
['dim2', 'dim0', 'dim1', 'aux0']
>>> d.get_keys('dim[1-9]')
['dim2', 'dim1']
```

has_key(*k*) → True if CFD has a key k, else False

insert_coordinate(*coord*, *dim_name_map=None*, *dim=False*, *aux=False*, *dimensions=None*, *space=None*, *key=None*)

Insert a new dimension or auxiliary coordinate to the space in place.

Parameters

coord [Coordinate] The new coordinate.

dim_name_map : dict, optional

dim [bool, optional] True if the new coordinate is to be a dimension coordinate.

aux [bool, optional] True if the new coordinate is to be an auxiliary coordinate.

dimensions [list] The ordered dimensions of the new coordinate. Ignored if the coordinate is a dimension coordinate. Required if the coordinate is an auxiliary coordinate.

space [Space, optional] Provide a space which contains both the new coordinate and its transforms, thus enabling transforms of the new coordinate to be included. By default, transforms of the new coordinate are not included.

key [str, optional] The identifier for the new coordinate. By default a unique identifier will be generated.

Returns None

Examples

```
>>>
```

new_auxiliary()

Return a new

new_dimension()

Return a new

new_transform()

Return a new

remove_coordinate(*key*)

Remove a coordinate from the space in place.

Parameters

key [str] The coordinate's identifier.

Returns None

Examples

```
>>> s.remove_coordinate('dim0')
>>> s.remove_coordinate('aux1')
```

squeeze(*dim*)

Remove a size 1 dimension from the space in place.

If the dimension has a dimension coordinate then it is removed, as are 1-d auxiliary coordinates and cell measures which span the dimension. The dimension is squeezed from multidimensional auxiliary coordinates and cell measures if they span it.

The dimension is not squeezed from the field's data array if it spans it, therefore the field's data array may need to be squeezed concurrently.

Parameters

dim [str] The identifier of the dimension to remove.

Returns None

Examples

```
>>> s.dimension_sizes
{'dim0': 12, 'dim1': 73, 'dim2': 1}
>>> s.dimensions
{'data': ['dim0', 'dim1', 'dim2'],
 'aux0': ['dim1', 'dim2'],
 'aux1': ['dim2', 'dim1'],
 'dim0': ['dim0'],
 'dim1': ['dim1'],
 'dim2': ['dim2'],
 'cm0' : ['dim1', 'dim2']}
>>> s.squeeze('dim2')
>>> s.dimension_sizes
{'dim0': 12, 'dim1': 73}
>>> s.dimensions
{'data': ['dim0', 'dim1', 'dim2'],
 'aux0': ['dim1'],
 'aux1': ['dim1'],
 'dim0': ['dim0'],
 'dim1': ['dim1'],
 'cm0' : ['dim1']}
```

dimension_sizes = None

The sizes of the space's dimensions.

Examples

```
>>> s.dimension_sizes
{'dim2': 96,
 'dim1': 73,
 'dim0': 1}
```

dimensions = None

The dimensions of each of the space's components and of the its field's data array.

```
>>> s.dimensions
{'data': ['dim0', 'dim1', 'dim2'],
 'aux0': ['dim1', 'dim2'],
 'aux1': ['dim2', 'dim1'],
 'dim0': ['dim0'],
 'dim1': ['dim1'],
 'dim2': ['dim2'],
 'cm0' : ['dim1', 'dim2']}
```

transforms = None

The space's transform constructs.

```
>>> s.transforms
{'trans0': <CF Transform: ocean_sigma_z_coordinate>
>>> type(s.transforms)
'cf.utils.CfDict'
```

8.2.8 cf.Transform

class cf.**Transform**(**args*, ***kwargs*)

Bases: cf.utils.CfDict

A CF transform construct.

The named parameters and their values of the transformation (i.e. the transformation's mappings) comprise the object's key-value pairs.

A transformation is equivalent to either a netCDF(CF) 'formula_terms' or 'grid_mapping' property. The latter is identified by the presence of the 'grid_mapping_name' key which contains the mapping's name.

In the 'formula_terms' case, a mapping to a coordinate (as opposed to another field) uses the coordinate's space key name as a pointer rather than a copy of the coordinate itself.

Examples

```
>>> t
<CF Transform: atmosphere_sigma_coordinate>
>>> print t.dump()
atmosphere_sigma_coordinate transform
-------------------------------------
Transform['ps'] = <CF Field: surface_air_pressure(73, 96)>
Transform['ptop'] = 0.05
Transform['sigma'] = 'dim0'

>>> t
<CF Transform: rotated_latitude_longitude>
>>> print t.dump()
rotated_latitude_longitude transform
------------------------------------
Transform['grid_mapping_name'] = 'rotated_latitude_longitude'
Transform['grid_north_pole_latitude'] = 33.67
Transform['grid_north_pole_longitude'] = 190.0
```

Initialization

Parameters

args, kwargs Keys and values are initialized exactly as for a built-in dict. Keys are transform parameter names (such as grid_north_pole_latitude or sigma) with appropriate values of string, numeric, Coordinate or Field types.

copy()

Return a deep copy.

Equivalent to `copy.deepcopy(d)`.

Returns

out [] The deep copy.

Examples

```
>>> d.copy()
```

dump(*id=None*)

Return a string containing a full description of the transform.

Parameters

id [str, optional] Set the common prefix of variable component names. By default the instance's class name is used.

Returns

out [str] A string containing the description.

Examples

```
>>> x = t.dump()
>>> print t.dump()
>>> print t.dump(id='transform1')
```

equals(*other*, *rtol=None*, *atol=None*, *traceback=False*)

True if two instances are logically equal, False otherwise.

Parameters

other [] The object to compare for equality.

atol [float, optional] The absolute tolerance for all numerical comparisons, By default the value returned by the *ATOL* function is used.

rtol [float, optional] The relative tolerance for all numerical comparisons, By default the value returned by the *RTOL* function is used.

traceback [bool, optional] If True then print a traceback highlighting where the two instances differ.

Returns

out [bool] Whether or not the two instances are equal.

Examples

get_keys(*regex=None*)
Return a list of the key names which match a regular expression.

Parameters

regex [str, optional] The regular expression with which to identify key names. By default all keys names are returned.

Returns

out [list] A list of key names.

Examples

```
>>> d.keys()
['dim2', 'dim0', 'dim1', 'aux0', 'cm0']
>>> d.get_keys()
['dim2', 'dim0', 'dim1', 'aux0', 'cm0']
>>> d.get_keys('dim')
['dim2', 'dim0', 'dim1']
>>> d.get_keys('^aux|^dim')
['dim2', 'dim0', 'dim1', 'aux0']
>>> d.get_keys('dim[1-9]')
['dim2', 'dim1']
```

has_key(*k*) → True if CFD has a key k, else False

hash()
Return a hash value for the transform.

Returns

out [int] The hash value.

Raises

ValueError: If the transform is not a grid mapping.

Examples

```
>>> t.hash()
5768338254506892753
```

is_formula_terms
True if the transform is a formula_terms.

Examples

```
>>> t
<CF Transform: rotated_latitude_longitude>
>>> t.is_formula_terms
False
```

is_grid_mapping
True if the transform is a grid_mapping.

Examples

```
>>> t
<CF Transform: rotated_latitude_longitude>
>>> t.is_grid_mapping
True
```

name = None
The identifying name of the transformation.

Examples

```
>>> t.name
'atmosphere_hybrid_sigma_pressure_coordinate'
>>> t.name = 'rotated_latitude_longitude'
```

8.2.9 cf.Units

class cf.**Units**(*units=None*, *calendar=None*, *format=None*, *names=None*, *definition=None*, *_ut_unit=None*)
Bases: object

Store, combine and compare physical units and convert numeric values to different units.

Units are as defined in UNIDATA's Udunits-2 package, with a few exceptions for greater consistency with the CF conventions namely support for CF calendars and new units definitions.

Modifications to the standard Udunits database

Whilst a standard Udunits-2 database may be used, greater consistency with CF is achieved by using a modified database. The following units are either new to, modified from, or removed from the standard Udunits-2 database (version 2.1.24):

Unit name	Symbol	Definition	Status
practical_salinity_unit	psu	1e-3	New unit
level		1	New unit
sigma_level		1	New unit
layer		1	New unit
decibel	dB	1	New unit
bel		10 dB	New unit
sverdrup	Sv	1e6 m3 s-1	Added symbol
sievert		J kg-1	Removed symbol

Plural forms of the new units' names are allowed, such as practical_salinity_units.

The modified database is in the *udunits* subdirectory of the *etc* directory found in the same location as this module.

Accessing units

Units may be set, retrieved and deleted via the *units* attribute. Its value is a string that can be recognized by UNIDATA's Udunits-2 package, with the few exceptions given in the CF conventions.

```
>>> u = Units('m s-1')
>>> u
<Cf Units: 'm s-1'>
>>> u.units = 'days since 2004-3-1'
>>> u
<CF Units: days since 2004-3-1>
```

Equality and equivalence of units

There are methods for assessing whether two units are equivalent or equal. Two units are equivalent if numeric values in one unit are convertible to numeric values in the other unit (such as `kilometres` and `metres`). Two units are equal if they are equivalent and their conversion is a scale factor of 1 and an offset of 0 (such as `kilometres` and `1000 metres`). Note that equivalence and equality are based on internally stored binary representations of the units, rather than their string representations.

```
>>> u = Units('m/s')
>>> v = Units('m s-1')
>>> w = Units('km.s-1')
>>> x = Units('0.001 kilometer.second-1')
>>> y = Units('gram')

>>> u.equivalent(v), u.equals(v),  u == v
(True, True, True)
>>> u.equivalent(w), u.equals(w)
(True, False)
>>> u.equivalent(x), u.equals(x)
(True, True)
>>> u.equivalent(y), u.equals(y)
(False, False)
```

Time and reference time units

Time units may be given as durations of time (*time units*) or as an amount of time since a reference time (*reference time units*):

```
>>> v = Units()
>>> v.units = 's'
>>> v.units = 'day'
>>> v.units = 'days since 1970-01-01'
>>> v.units = 'seconds since 1992-10-8 15:15:42.5 -6:00'
```

Note: It is recommended that the units `year` and `month` be used with caution, as explained in the following excerpt from the CF conventions: "The Udunits package defines a year to be exactly 365.242198781 days (the interval between 2 successive passages of the sun through vernal equinox). It is not a calendar year. Udunits includes the following definitions for years: a common_year is 365 days, a leap_year is 366 days, a Julian_year is 365.25 days, and a Gregorian_year is 365.2425 days. For similar reasons the unit `month`, which is defined to be exactly year/12, should also be used with caution."

Calendar

The date given in reference time units is associated with one of the calendars recognized by the CF conventions and may be set with the *calendar* attribute. However, as in the CF conventions, if the calendar is not set then, for the purposes of calculation and comparison, it defaults to the mixed Gregorian/Julian calendar as defined by Udunits:

```
>>> u = Units('days since 2000-1-1')
>>> u.calendar
AttributeError: Can't get 'Units' attribute 'calendar'
>>> v = Units('days since 2000-1-1')
>>> v.calendar = 'gregorian'
>>> v.equals(u)
True
```

Arithmetic with units

The following operators, operations and assignments are overloaded:

Comparison operators:

```
==, !=
```

Binary arithmetic operations:

```
+, -, *, /, pow(), **
```

Unary arithmetic operations:

```
-, +
```

Augmented arithmetic assignments:

```
+=, -=, *=, /=, **=
```

The comparison operations return a boolean and all other operations return a new units object or modify the units object in place.

```
>>> u = Units('m')
<CF Units: m>

>>> v = u * 1000
>>> v
<CF Units: 1000 m>

>>> u == v
False
>>> u != v
True

>>> u **= 2
>>> u
<CF Units: m2>
```

It is also possible to create the logarithm of a unit corresponding to the given logarithmic base:

```
>>> u = Units('seconds')
>>> u.log(10)
<CF Units: lg(re 1 s)>
```

Modifying data for equivalent units

Any numpy array or python numeric type may be modified for equivalent units using the *conform* static method.

```
>>> Units.conform(2, Units('km'), Units('m'))
2000.0

>>> import numpy
>>> a = numpy.arange(5.0)
>>> Units.conform(a, Units('minute'), Units('second'))
array([   0.,   60.,  120.,  180.,  240.])
>>> a
array([ 0.,  1.,  2.,  3.,  4.])
```

If the *inplace* keyword is True, then a numpy array is modified in place, without any copying overheads:

```
>>> Units.conform(a,
                  Units('days since 2000-12-1'),
                  Units('days since 2001-1-1'), inplace=True)
array([-31., -30., -29., -28., -27.])
>>> a
array([-31., -30., -29., -28., -27.])
```

Initialization

Parameters

units [str, optional] Set the new units from this string.

calendar [str, optional] Set the calendar for reference time units.

format [bool, optional] Format the string representation of the units in a standardized manner. See the *format* method.

names [bool, optional] Format the string representation of the units using names instead of symbols. See the *format* method.

definition [bool, optional] Format the string representation of the units using basic units. See the *format* method.

_ut_unit [int, optional] Set the new units from this Udunits binary unit representation. This should be an integer returned by a call to *ut_parse* function of Udunits. Overrides *units*, if set.

static conform(*x*, *from_units*, *to_units*, *inplace=False*)

Conform values in one unit to equivalent values in another, compatible unit. Returns the conformed values.

The values may either be a numpy array or a python numeric type. The returned value is of the same type, except that input integers are converted to floats (see the *inplace* keyword).

Parameters x : numpy.ndarray or python numeric

from_units [Units] The original units of *x*

to_units [Units] The units to which *x* should be conformed to.

inplace [bool, optional] If True and *x* is a numpy array then change it in place, creating no temporary copies, with one exception: If *x* is of integer type and the conversion is not null, then it will not be changed inplace and the returned conformed array will be of float type.

Returns

out [numpy.ndarray or python numeric] The modified numeric values.

Examples

```
>>> Units.conform(2, Units('km'), Units('m'))
2000.0

>>> import numpy
>>> a = numpy.arange(5.0)
>>> Units.conform(a, Units('minute'), Units('second'))
array([   0.,   60.,  120.,  180.,  240.])
>>> a
array([ 0.,  1.,  2.,  3.,  4.])

>>> Units.conform(a,
                  Units('days since 2000-12-1'),
                  Units('days since 2001-1-1'), inplace=True)
array([-31., -30., -29., -28., -27.])
>>> a
array([-31., -30., -29., -28., -27.])
```

Warning: Do not change the calendar of reference time units in the current version. Whilst this is possible, it will almost certainly result in an incorrect interpretation of the data or an error. Allowing the calendar to be changed is under development and will be available soon.

copy()

Return a deep copy.

Equivalent to `copy.deepcopy(u)`.

Returns

out : The deep copy.

Examples

```
>>> v = u.copy()
```

dump(*id=None*, *omit=()*)

equals(*other*, *rtol=None*, *atol=None*)

Return True if and only if numeric values in one unit are convertible to numeric values in the other unit and their conversion is a scale factor of 1.

Parameters

other [Units] The other units.

Returns

out [bool] True if the units are equal, False otherwise.

Examples

```
>>> u = Units(units='km')
>>> v = Units(units='1000m')
>>> u.equals(v)
True

>>> u = Units(units='m s-1')
>>> m = Units(units='m')
>>> s = Units(units='s')
>>> u.equals(m)
False
>>> u.equals(m/s)
True
>>> (m/s).equals(u)
True
```

Undefined units are considered equal:

```
>>> u = Units()
>>> v = Units()
>>> u.equals(v)
True
```

equivalent(*other*)

Returns True if and only if numeric values in one unit are convertible to numeric values in the other unit.

Parameters

other [Units] The other units.

Returns

out [bool] True if the units are equivalent, False otherwise.

Examples

```
>>> u = Units(units='m')
>>> v = Units(units='km')
>>> w = Units(units='s')

>>> u.equivalent(v)
True
>>> u.equivalent(w)
False

>>> u = Units(units='days since 2000-1-1')
>>> v = Units(units='days since 2000-1-1', calendar='366_day')
>>> w = Units(units='seconds since 1978-3-12')
```

```
>>> u.equivalent(v)
False
>>> u.equivalent(w)
True
```

format(*names=None*, *definition=None*)

Formats the string stored in the *units* attribute in a standardized manner. The *units* attribute is modified in place and its new value is returned.

Parameters

names [bool, optional] Use unit names instead of symbols.

definition [bool, optional] The formatted string is given in terms of basic-units instead of stopping any expansion at the highest level possible.

Returns

out [str] The formatted string.

Examples

```
>>> u = Units(units='W')
>>> u.units
'W'
>>> u.format()
>>> u.units
'W'
>>> u.format(names=True)
>>> u.units
'watt'
>>> u.format(definition=True)
>>> u.units
'm2.kg.s-3'
>>> u.format(names=True, definition=True)
'meter^2-kilogram-second^-3'
>>> u.format()
'W'

>>> u.units='dram'
>>> u.format(names=True)
'1.848345703125e-06 meter^3'
```

Formatting is also available during object initialization:

```
>>> u = Units(units='m/s', format=True)
>>> u.units
'm.s-1'

>>> u = Units(units='dram', names=True)
>>> u.units
'1.848345703125e-06 m3'

>>> u = Units(units='W', names=True, definition=True)
>>> u.units
'meter^2-kilogram-second^-3'
```

log(*base*)

Returns the logarithmic unit corresponding to the given logarithmic base.

Parameters

base [int or float] The logarithmic base.

Returns

out [Units] The logarithmic unit corresponding to the given logarithmic base.

Examples

```
>>> u = Units(units='W', names=True)
>>> u
<CF Units: watt>

>>> u.log(10)
<CF Units: lg(re 1 W)>
>>> u.log(2)
<CF Units: lb(re 1 W)>

>>> import math
>>> u.log(math.e)
<CF Units: ln(re 1 W)>

>>> u.log(3.5)
<CF Units: 0.798235600147928 ln(re 1 W)>
```

calendar

The calendar CF attribute for reference time units.

Note that deleting the calendar will adjust the reference time units for the default CF calendar ('gregorian'), if necessary.

Examples

```
>>> u = Units()
>>> u.calendar = 'gregorian'
>>> u.calendar
'gregorian'

>>> u = Units(calendar='365_day')
>>> u.calendar
'365_day'
>>> del u.calendar
```

is_latitude

True if the units are latitude units, i.e. if the *units* attribute is 'degrees_north', false otherwise.

Examples

```
>>> u = Units(units='degrees_north')
>>> u.is_latitude
True

>>> u = Units(units='degrees')
>>> u.is_latitude
False

>>> u = Units(units='degrees_east')
>>> u.is_latitude
False
```

is_longitude

True if the units are longitude units, i.e. if the *units* attribute is 'degrees_east', false otherwise.

Examples

```
>>> u = Units(units='degrees_east')
>>> u.is_longitude
True

>>> u = Units(units='degrees')
>>> u.is_longitude
False
```

```
>>> u = Units(units='degrees_north')
>>> u.is_longitude
False
```

is_pressure

True if the units are pressure units, false otherwise.

Examples

```
>>> u = Units(units='bar')
>>> u.is_pressure
True

>>> u = Units(units='hours since 2100-1-1', calendar='noleap')
>>> u.is_pressure
False
```

is_reftime

True if the units are reference time units, false otherwise.

Note that time units are not reference time units and that the existence of the *calendar* attribute on its own is not sufficient for identifying the presence of reference time units.

Examples

```
>>> u = Units(units='days since 1989-1-1')
>>> u.is_reftime
True

>>> u = Units(units='hours since 2100-1-1', calendar='noleap')
>>> u.is_reftime
True

>>> u = Units(units='kg')
>>> u.is_reftime
False

>>> u = Units(units='hours')
>>> u.is_reftime
False

>>> u = Units(calendar='360_day')
>>> u.is_reftime
False
```

is_time

True if the units are time units, False otherwise.

Note that reference time units are not time units.

Examples

```
>>> u = Units(units='days')
>>> u.is_time
True

>>> u = Units(units='hours since 2100-1-1', calendar='noleap')
>>> u.is_time
False

>>> u = Units(calendar='360_day')
>>> u.is_time
False
```

```
>>> u = Units(units='kg')
>>> u.is_time
False
```

units

The units string. May be any string allowed by the *units* attribute in the CF conventions.

Examples

```
>>> u = Units()
>>> u.units = 'kg'
>>> u.units
'kg'

>>> u = Units(units='percent')
>>> u.units
'percent'
>>> del u.units
```

8.3 Miscellaneous classes

Comparison	Create an object for storing a comparison expression.
CoordinateList	An ordered sequence of coordinates stored in a list-like object.
FieldList	An ordered sequence of fields stored in a list-like object.

8.3.1 cf.Comparison

class cf.**Comparison**(*relation*, *value*, *units=None*)

Bases: object

Create an object for storing a comparison expression.

A comparison expression comprises a relation (such as 'less than') and a value (any object capable of being related to with the standard comparison operators). These are stored in the *relation* and *value* attributes respectively.

The comparison is evaluated for an arbitrary object with the *evaluate* method.

Valid relations are:

Relation	Description
'lt'	Strictly less than
'le'	Less than or equal
'gt'	Strictly greater than
'ge'	Greater than or equal
'eq'	Equals (within a tolerance)
'ne'	Not equal to (within a tolerance)
'inside'	Inside a given range (range bounds included)
'outside'	Outside a given range (range bounds excluded)

As a convenience, for each relation in the above list there is an identically named function which returns the appropriate Comparison object.

Examples

```
>>> c = cf.Comparison('le', 5)
>>> c.evaluate(4)
True
>>> c.evaluate(5)
True
```

```
>>> c = lt(5)
>>> c.evaluate(4)
True

>>> c = cf.Comparison('inside', (1,2))
>>> a = numpy.arange(4)
>>> c.evaluate(a)
array([False,  True,  True, False], dtype=bool)
>>> (a >= 1) & (a <= 2)
array([False,  True,  True, False], dtype=bool)
```

Initialization

Parameters

relation [str] The comparison operator type.

value [object] The value on the right hand side of the comparison operation.

units [str or Units, optional] The units of *value*. By default, the same units as the left hand side of the comparison operation are assumed.

copy()

Return a deep copy.

Equivalent to `copy.deepcopy(c)`.

Returns

out : The deep copy.

Examples

```
>>> c.copy()
```

dump(*id=None*)

Return a string containing a full description of the instance.

Parameters

id [str, optional] Set the common prefix of component names. By default the instance's class name is used.

Returns

out [str] A string containing the description.

Examples

```
>>> x = c.dump()
>>> print c.dump()
>>> print c.dump(id='comparison1')
```

evaluate(*x*)

Evaluate the comparison expression for a given object.

Parameters

x [object] The object for the left hand side of the comparison expression.

Returns

out [] The output of the comparison given by the *relation* attribute with *x* on the left hand side and the *value* attribute on the right hand side.

Examples

```
>>> c = cf.Comparison('lt', 5.5)
>>> c.evaluate(6)
False
```

```
>>> c = cf.Comparison('inside', (1,2))
>>> array = numpy.arange(4)
>>> array
array([0, 1, 2, 3])
>>> c.evaluate(array)
array([False,  True,  True, False], dtype=bool)
```

8.3.2 cf.CoordinateList

class `cf.`**`CoordinateList`**(*sequence=()*)

Bases: `cf.variablelist.VariableList`

An ordered sequence of coordinates stored in a list-like object.

In some contexts, whether an object is a coordinate or a coordinate list is not known and does not matter. So to avoid ungainly type testing, some aspects of the CoordinateList interface are shared by a coordinate and vice versa.

Any attribute or method belonging to a coordinate may be used on a coordinate list and will be applied independently to each element.

Just as it is straight forward to iterate over the coordinates in a coordinate list, a coordinate will behave like a single element coordinate list in iterative and indexing contexts.

Initialization

Parameters

sequence [iterable, optional] Define a new list with these elements.

`copy`()

Return a deep copy.

Equivalent to `copy.deepcopy(s)`.

Returns

out [] The deep copy.

Examples

```
>>> s.copy()
```

`count`(*value*)

Return the number of occurrences of a given value.

Parameters

value : The value to count.

Returns

out [int] The number of occurrences of *value*.

Examples

```
>>> s
[1, 2, 3, 2, 4, 2]
>>> s.count(1)
1
>>> s.count(2)
3
```

`delattr`(*attr*)

Delete a public attribute from each element of the list of variables.

`dump`(**arg*, ***kwargs*)

Return a string containing the full descriptions of each variable in the list.

equals(*other*, *rtol=None*, *atol=None*, *traceback=False*)
True if two instances are logically equal, False otherwise.

Parameters

other [] The object to compare for equality.

atol [float, optional] The absolute tolerance for all numerical comparisons, By default the value returned by the *ATOL* function is used.

rtol [float, optional] The relative tolerance for all numerical comparisons, By default the value returned by the *RTOL* function is used.

traceback [bool, optional] If True then print a traceback highlighting where the two instances differ.

Returns

out [bool] Whether or not the two instances are equal.

extract(**args*, ***kwargs*)
Return the elements which match the given conditions.

The match conditions are passed to each element's *match* method in turn.

Parameters

args, kwargs : As for the variable's *match* method.

Returns

out : A new list containing the matching elements.

Examples

```
>>> f
[<>
 <>]
>>> f.extract(attr={'standard_name': 'air_temperature'})
[<>]
```

getattr(**args*)
Return a built-in list of the public attributes of each element of the list of variables.

hasattr(**args*)
Return a built-in list of the public attributes of each element of the list of variables.

index(*value*, *start=0*, *stop=None*)
Return the first index of a given value.

Parameters value :

start : int, optional

stop : int, optional

Returns out : int

Raises

ValueError : If the given value is not in the list.

Examples

```
>>> s
[1, 2, 3, 2, 4, 2]
>>> s.index(1)
1
>>> s.index(2, start=2)
3
>>> s.index(2, start=2, stop=4)
3
```

insert(*index*, *object*)
Insert an object before the given index in place.

Parameters index : int

object :

Returns None

Examples

```
>>> s
[1, 2, 3]
>>> s.insert(1, 'A')
>>> s
[1, 'A', 2, 3]
```

match(**args*, ***kwargs*)
Return a list of booleans showing which elements match the given conditions.

The match conditions are passed to each element's *match* method in turn.

Parameters

args, kwargs : As for the variable's *match* method.

Returns

out [list] A built-in list of booleans showing which elements match the conditions.

Examples

```
>>> f
[<>
 <>]
>>> f.match(attr={'standard_name': 'air_temperature'})
[True, False]
```

name(**arg*, ***kwargs*)
Return a built-in list of the names of each element of the list of variables.

setattr(*attr*, *value*)
Set a public attribute from each element of the list of variables.

subset
Subset each variable in the list, returning a new list of variables.

Examples

```
>>> vl
[<CF Variable: air_temperature(73, 96)>,
 <CF Variable: air_temperature(73, 96)>]
>>> vl.subset[0,0]
[<CF Variable: air_temperature(1,1)>,
 <CF Variable: air_temperature(1,1)>]
```

8.3.3 cf.FieldList

class cf.**FieldList**(*sequence=()*)
Bases: cf.variablelist.VariableList

An ordered sequence of fields stored in a list-like object.

In some contexts, whether an object is a field or a field list is not known and does not matter. So to avoid ungainly type testing, some aspects of the FieldList interface are shared by a field and vice versa.

Any attribute or method belonging to a field may be used on a field list and will be applied independently to each element.

Just as it is straight forward to iterate over the fields in a field list, a field will behave like a single element field list in iterative and indexing contexts.

Initialization

Parameters

sequence [iterable, optional] Define a new list with these elements.

coord(**args*, ***kwargs*)
Apply the *coord* method to each element of the list.

Note that a coordinate list is returned (as opposed to a field list).

Parameters

args, kwargs : As for a field's *coord* method.

Returns out : CoordinateList

copy()
Return a deep copy.

Equivalent to `copy.deepcopy(s)`.

Returns

out [] The deep copy.

Examples

```
>>> s.copy()
```

count(*value*)
Return the number of occurrences of a given value.

Parameters

value : The value to count.

Returns

out [int] The number of occurrences of *value*.

Examples

```
>>> s
[1, 2, 3, 2, 4, 2]
>>> s.count(1)
1
>>> s.count(2)
3
```

delattr(*attr*)
Delete a public attribute from each element of the list of variables.

dump(**arg*, ***kwargs*)
Return a string containing the full descriptions of each variable in the list.

equals(*other*, *rtol=None*, *atol=None*, *traceback=False*)
True if two instances are logically equal, False otherwise.

Parameters

other [] The object to compare for equality.

atol [float, optional] The absolute tolerance for all numerical comparisons, By default the value returned by the *ATOL* function is used.

rtol [float, optional] The relative tolerance for all numerical comparisons, By default the value returned by the *RTOL* function is used.

traceback [bool, optional] If True then print a traceback highlighting where the two instances differ.

Returns

out [bool] Whether or not the two instances are equal.

extract(**args*, ***kwargs*)

Return the elements which match the given conditions.

The match conditions are passed to each element's *match* method in turn.

Parameters

args, kwargs : As for the variable's *match* method.

Returns

out : A new list containing the matching elements.

Examples

```
>>> f
[<>
 <>]
>>> f.extract(attr={'standard_name': 'air_temperature'})
[<>]
```

getattr(**args*)

Return a built-in list of the public attributes of each element of the list of variables.

hasattr(**args*)

Return a built-in list of the public attributes of each element of the list of variables.

index(*value*, *start=0*, *stop=None*)

Return the first index of a given value.

Parameters value :

start : int, optional

stop : int, optional

Returns out : int

Raises

ValueError : If the given value is not in the list.

Examples

```
>>> s
[1, 2, 3, 2, 4, 2]
>>> s.index(1)
1
>>> s.index(2, start=2)
3
>>> s.index(2, start=2, stop=4)
3
```

insert(*index*, *object*)

Insert an object before the given index in place.

Parameters index : int

object :

Returns None

Examples

```
>>> s
[1, 2, 3]
>>> s.insert(1, 'A')
>>> s
[1, 'A', 2, 3]
```

match(**args*, ***kwargs*)
Return a list of booleans showing which elements match the given conditions.

The match conditions are passed to each element's *match* method in turn.

Parameters

args, kwargs : As for the variable's *match* method.

Returns

out [list] A built-in list of booleans showing which elements match the conditions.

Examples

```
>>> f
[<>
 <>]
>>> f.match(attr={'standard_name': 'air_temperature'})
[True, False]
```

name(**arg*, ***kwargs*)
Return a built-in list of the names of each element of the list of variables.

setattr(*attr*, *value*)
Set a public attribute from each element of the list of variables.

squeeze(**args*, ***kwargs*)
Apply the squeeze method to each element of the list.

Parameters

args, kwargs : As for a field's *squeeze* method.

Returns out : FieldList

unsqueeze(**args*, ***kwargs*)
Apply the unsqueeze method to each element of the list.

Parameters

args, kwargs : As for a field's *unsqueeze* method.

Returns out : FieldList

subset
Slice each field in the list, returning a new list of fields. Slicing by indices or by coordinate values are both allowed.

Examples

```
>>> fl
[<CF Variable: air_temperature(73, 96)>,
 <CF Variable: air_temperature(73, 96)>]
>>> fl.subset[0,0]
[<CF Variable: air_temperature(1,1)>,
 <CF Variable: air_temperature(1,1)>]
>>> fl.slice(longitude=0, latitude=0)
[<CF Variable: air_temperature(1,1)>,
 <CF Variable: air_temperature(1,1)>]
```

8.4 Data component classes

Partition	A partition.
PartitionArray	An N-dimensional partition array.
FileArray	A numpy array stored on disk in a temporary file.

8.4.1 cf.Partition

class cf.**Partition**(***kwargs*)

Bases: object

A partition.

Initialization

Parameters

data [numpy array-like] The data for the partition. May be a numpy array or any object which shares the numpy __getitem__, dtype, ndim, size, and shape interface

direction [dict or bool] The direction of each dimension of the partition's data. It is a boolean if the partition's data is a scalar array, otherwise it is a dictionary keyed by the dimensions' identities as found in *order*.

location [list] The location of the partition's data in the master array.

order [list] The identities of the dimensions of the partition's data. If the partition's data a scalar array then it is an empty list.

part [list] The subspace of the partition's data to be returned when it is accessed. If the partition's data is to be returned complete then *part* may be an empty list.

shape [list] The shape of the partition's data as a subspace of the master array. If the master array is a scalar array then *shape* is an empty list.

Units [Units] The units of the partition's data.

Examples

```
>>> p = Partition(data      = numpy.arange(20).reshape(2,5,1),
                  direction = {'dim0', True, 'dim1': False, 'dim2': True},
                  location  = [(0, 6), (1, 3), (4, 5)],
                  order     = ['dim1', 'dim0', 'dim2'],
                  shape     = [5, 2, 1],
                  Units     = cf.Units('K'),
                  part      = [],

>>> p = Partition(data      = numpy.arange(20).reshape(2,5,1),
                  direction = {'dim0', True, 'dim1': False, 'dim2': True},
                  location  = [(0, 6), (1, 3), (4, 5)],
                  order     = ['dim1', 'dim0', 'dim2'],
                  shape     = [5, 2, 1],
                  Units     = cf.Units('K'),
                  part      = [slice(None, None, -1), [0,1,3,4], slice(None)],

>>> p = Partition(data      = numpy.array(4),
                  direction = True,
                  location  = [(4, 5)],
                  order     = ['dim1'],
                  shape     = [1],
                  Units     = cf.Units('K'),
                  part      = [],
```

close(*save=False*)

Close the partition.

Closing the partition is important for memory management. The partition should always be closed after it is conformed to prevent memory leaks.

Closing the partition does one of three things, depending on the values of the partition's *_original* and *_save* attributes and on the *save* parameter: * Nothing. * Stores the data in a temporary file. * Reverts the partition to a previous state.

Parameters

save [bool, optional] If True and the partition is not to be reverted to a previous state then force its data to be stored in a temporary file.

Returns None

Examples

```
>>> p.close()
```

conform(*data_order*, *data_direction*, *units*, *save=False*, *revert_to_file=False*)

After a partition has been conformed, the partition must be closed (with the *close* method) before another partition is conformed, otherwise a memory leak could occur. For example:

```
>>> order          = partition_array.order
>>> direction      = partition_array.direction
>>> units          = partition_array.units
>>> save           = partition_array.save_to_disk()
>>> revert_to_file = True
>>> for partition in partition_array.flat():
...
...     # Conform the partition
...     partition.conform(order, direction, units, save, revert_to_file)
...
...     # [ Some code to operate on the conformed partition ]
...
...     # Close the partition
...     partition.close()
...
...     # Now move on to conform the next partition
...
>>>
```

Parameters order : list

direction : dict

units : Units

save [bool, optional]

- If False then the conformed partition's data is to be kept in memory when the partition's *close* method is called.
- If True and *revert_to_file* is False then the conformed partition's data will be to be saved to a temporary file on disk when the partition's *close* method is called.
- If True and *revert_to_file* is True and the pre-conformed partition's data was in memory then the conformed partition's data will be saved to a temporary file on disk when the partition's *close* method is called.
- If True and *revert_to_file* is True and the pre-conformed partition's data was on disk then the file pointer will be reinstated when the partition's *close* method is called.

revert_to_file [bool, optional]

- If False and *save* is True then the conformed partition's data will be saved to a temporary file on disk when the partition's *close* method is called.
- If True and *save* is True and the pre-conformed partition's data was on disk then the file pointer will be reinstated when the partition's *close* method is called.
- Otherwise does nothing.

Returns

out [numpy array] The partition's conformed data as a numpy array. The same numpy array is stored as an object identity by the partition's *data* attribute.

copy()
Return a deep copy.

Equivalent to `copy.deepcopy(p)`.

Returns

out : A deep copy.

Examples

```
>>> q = p.copy()
```

new_part(*indices*, *dim2position*, *data_direction*)
Return the *part* attribute updated for new indices.

Does not change the partition in place.

Parameters indices : list

dim2position : dict

data_direction : dict

Returns out : list

Examples

```
>>> p.part = p.new_part(indices, dim2position, data_direction)
```

to_disk()
Store the partition's data in a temporary file on disk in place.

Assumes that the partition's data is currently in memory, but this is not checked.

Returns None

Examples

```
>>> p.to_disk()
```

update_from(*other*)
Completely update the partition with another partition's attributes in place.

The updated partition is always independent of the other partition.

Parameters other : Partition

Returns None

Examples

```
>>> p.update_from(q)
```

in_memory
True if and only if the partition's data is in memory as opposed to on disk.

Examples

```
>>> p.in_memory
False
```

indices
The indices of the master array which correspond to this partition's data.

Returns

out [tuple] A tuple of slice objects or, if the partition's data is a scalar array, an empty tuple.

Examples

```
>>> p.data.shape
(5, 7)
>>> p.indices
(slice(0, 5), slice(2, 9))

>>> p.data.shape
()
>>> p.indices
()
```

is_scalar
True if and only if the partition's data is a scalar array.

Examples

```
>>> p.data.ndim
0
>>> p.is_scalar
True

>>> p.data.ndim
1
>>> p.is_scalar
False
```

on_disk
Return True if the partition's data is on disk as opposed to in memory.

Examples

```
>>> p.on_disk
True
```

8.4.2 cf.PartitionArray

class cf.**PartitionArray**(*array*, ***kwargs*)
Bases: cf.utils.CfList

An N-dimensional partition array.

Initialization

Parameters

array [list] A list of Partition objects.

kwargs : Set attributes specified by the keywords and their values. Values are not deep copied.

Examples

```
>>> pa = PartitionArray(
        [Partition(location  = [(0,n) for n in shape],
                   shape     = shape[:],
                   order     = order[:],
                   direction = copy(direction),
                   Units     = units.copy(),
                   part      = [],
                   data      = data)
         ],
        order      = order,
        shape      = shape,
        size       = data.size,
        Units      = units.copy(),
        ndim       = ndim,
        direction  = direction,
        adims      = [],
        _FillValue = _FillValue)
```

add_partitions(*extra_boundaries*, *adim*, *existing_boundaries=None*)

change_dimension_names(*dim_name_map*)

dim_name_map should be a dictionary which maps each dimension names in self.order to its new dimension name. E.g. {'dim0':'dim1', 'dim1':'dim0'}

copy(*_copy_attr=True*)

Return a deep copy.

Do not set the *_copy_attr* parameter. It is for internal use only.

Equivalent to `copy.deepcopy(pa)`.

Returns

out : The deep copy.

Examples

```
>>> pa.copy()
```

count(*value*)

Return the number of occurrences of a given value.

Parameters

value : The value to count.

Returns

out [int] The number of occurrences of *value*.

Examples

```
>>> s
[1, 2, 3, 2, 4, 2]
>>> s.count(1)
1
>>> s.count(2)
3
```

expand_dims(*adim*)

flat()

Return a flat iterator over the Partition objects in the partition array.

Returns

out [generator] An iterator over the elements of the partition array.

Examples

```
>>> type(pa.flat())
<generator object flat at 0x145a0f0>
>>> for partition in pa.flat():
...     print partition.Units
```

index(*value*, *start=0*, *stop=None*)

Return the first index of a given value.

Parameters value :

start : int, optional

stop : int, optional

Returns out : int

Raises

ValueError : If the given value is not in the list.

Examples

```
>>> s
[1, 2, 3, 2, 4, 2]
>>> s.index(1)
1
>>> s.index(2, start=2)
3
>>> s.index(2, start=2, stop=4)
3
```

info(*attr*)

insert(*index*, *object*)

Insert an object before the given index in place.

Parameters index : int

object :

Returns None

Examples

```
>>> s
[1, 2, 3]
>>> s.insert(1, 'A')
>>> s
[1, 'A', 2, 3]
```

partition_boundaries()

ravel()

Return a flattened partition array as a built-in list.

Returns

out [list of Partitions] A list containing the flattened partition array.

Examples

```
>>> x = partitions.ravel()
>>> type(x)
list
```

rollaxis(*axis*, *start=0*)

Roll the specified aggregating dimension backwards,in place until it lies in a given position.

This does not change the master array.

Parameters

axis [int] The axis to roll backwards. The positions of the other axes do not change relative to one another.

start [int, optional] The axis is rolled until it lies before this position. The default, 0, results in a "complete" roll.

Returns None

Examples

```
>>> pa.rollaxis(2)
>>> pa.rollaxis(2, start=1)
```

save_to_disk(*itemsize=None*)

Parameters itemsize : int, optional

Returns out : bool

set_location_map()

Recalculate the *location* attribute of each Partition object in the partition array in place.

Examples

```
>>> pa.set_location_map()
```

to_disk()

Store each partition's data on disk in place.

There is no change to partitions with data that are already on disk.

Returns None

Examples

```
>>> pa.to_disk()
```

to_memory(*regardless=False*)

Store each partition's data in memory in place if the master array is smaller than the chunk size.

There is no change to partitions with data that are already in memory.

Parameters

regardless [bool, optional] If True then store all partitions' data in memory regardless of the size of the master array. By default only store all partitions' data in memory if the master array is smaller than the chunk size.

Returns None

Examples

```
>>> pa.to_memory()
>>> pa.to_memory(True)
```

transpose(*axes*)

Permute the aggregating dimensions of the partition array in-place.

This does not change the master array.

Parameters

axes [sequence of ints] Permute the axes according to the values given.

Returns None

Examples

```
>>> pa.transpose((2,0,1))
```

dtype
Numpy data-type of the master array.

When there is more than one partition in the partition array, the normal data-type coercion rules apply. For example, if the partitions have data-types 'int32' and 'float32' then the master array's data-type will be 'float32'.

Examples

```
>>> pa.dtype
dtype('float64')
```

is_scalar
True if the master array is a 0-d scalar array.

Examples

```
>>> pa.ndim
0
>>> pa.is_scalar
True

>>> pa.ndim >= 1
True
>>> pa.is_scalar
False
```

npartitions
The number of partitions in the partition array.

Logically equivalent to `len(pa.ravel())`.

Examples

```
>>> pa.npartitions
7
```

8.4.3 cf.Filearray

class `cf.`**`FileArray`**(*array*)
Bases: `object`

A numpy array stored on disk in a temporary file.

Initialization

Parameters

array [numpy array] The array to be stored on disk in a temporary file.

Examples

```
>>> f = FileArray(numpy.ma.array([1, 2, 3, 4, 5]))
```

copy()
Return a deep copy.

Equivalent to `copy.deepcopy(f)`.

Returns

out : A deep copy.

Examples

```
>>> f.copy()
```

dtype
Numpy data-type of the array.

Examples

```
>>> f.dtype
dtype('float64')
```

ndim
Number of dimensions in the array.

Examples

```
>>> f.shape
(73, 96)
>>> f.ndim
2
```

shape
Tuple of the array's dimension sizes.

Examples

```
>>> f.shape
(73, 96)
```

size
Number of elements in the array.

Examples

```
>>> f.shape
(73, 96)
>>> f.size
7008
```

8.5 Base classes

CfDict	A dictionary-like object with attributes.
CfList	A list-like object with attributes.
Variable	Base class for storing a data array with metadata.
VariableList	An ordered sequence of variables stored in a list-like object.

8.5.1 cf.CfDict

class cf.**CfDict**(**args*, ***kwargs*)
Bases: _abcoll.MutableMapping

A dictionary-like object with attributes.

Initialization

Parameters

args, kwargs : Keys and values are initialized exactly as for a built-in dict.

copy()
Return a deep copy.

Equivalent to copy.deepcopy(d).

Returns

out [] The deep copy.

Examples

```
>>> d.copy()
```

equals(*other*, *rtol=None*, *atol=None*, *traceback=False*)
True if two instances are logically equal, False otherwise.

Parameters

other [] The object to compare for equality.

atol [float, optional] The absolute tolerance for all numerical comparisons, By default the value returned by the *ATOL* function is used.

rtol [float, optional] The relative tolerance for all numerical comparisons, By default the value returned by the *RTOL* function is used.

traceback [bool, optional] If True then print a traceback highlighting where the two instances differ.

Returns

out [bool] Whether or not the two instances are equal.

Examples

get_keys(*regex=None*)
Return a list of the key names which match a regular expression.

Parameters

regex [str, optional] The regular expression with which to identify key names. By default all keys names are returned.

Returns

out [list] A list of key names.

Examples

```
>>> d.keys()
['dim2', 'dim0', 'dim1', 'aux0', 'cm0']
>>> d.get_keys()
['dim2', 'dim0', 'dim1', 'aux0', 'cm0']
>>> d.get_keys('dim')
['dim2', 'dim0', 'dim1']
>>> d.get_keys('^aux|^dim')
['dim2', 'dim0', 'dim1', 'aux0']
>>> d.get_keys('dim[1-9]')
['dim2', 'dim1']
```

has_key(*k*) → True if CFD has a key k, else False

8.5.2 cf.CfList

class cf.**CfList**(*sequence=()*)
Bases: _abcoll.MutableSequence

A list-like object with attributes.

Initialization

Parameters

sequence [iterable, optional] Define a new list with these elements.

copy()
Return a deep copy.

Equivalent to `copy.deepcopy(s)`.

Returns

out [] The deep copy.

Examples

```
>>> s.copy()
```

count(*value*)
Return the number of occurrences of a given value.

Parameters

value : The value to count.

Returns

out [int] The number of occurrences of *value*.

Examples

```
>>> s
[1, 2, 3, 2, 4, 2]
>>> s.count(1)
1
>>> s.count(2)
3
```

equals(*other*, *rtol=None*, *atol=None*, *traceback=False*)
True if two instances are logically equal, False otherwise.

Parameters

other [] The object to compare for equality.

atol [float, optional] The absolute tolerance for all numerical comparisons, By default the value returned by the *ATOL* function is used.

rtol [float, optional] The relative tolerance for all numerical comparisons, By default the value returned by the *RTOL* function is used.

traceback [bool, optional] If True then print a traceback highlighting where the two instances differ.

Returns

out [bool] Whether or not the two instances are equal.

index(*value*, *start=0*, *stop=None*)
Return the first index of a given value.

Parameters value :

start : int, optional

stop : int, optional

Returns out : int

Raises

ValueError : If the given value is not in the list.

Examples

```
>>> s
[1, 2, 3, 2, 4, 2]
>>> s.index(1)
1
>>> s.index(2, start=2)
3
>>> s.index(2, start=2, stop=4)
3
```

insert(*index*, *object*)
Insert an object before the given index in place.

Parameters index : int

object :

Returns None

Examples

```
>>> s
[1, 2, 3]
>>> s.insert(1, 'A')
>>> s
[1, 'A', 2, 3]
```

8.5.3 cf.Variable

class cf.**Variable**(*attributes={}*)
Bases: object

Base class for storing a data array with metadata.

A variable contains a data array and metadata comprising properties to describe the physical nature of the data.

All components of a variable are optional.

Initialization

Parameters

attributes [dict, optional] Initialize a new instance with attributes from the dictionary's key/value pairs. Values are deep copied.

binary_mask()
return new variable containing binary mask

chunk(*chunksize=None*, *extra_boundaries=[]*, *adim=[]*)
Partition the data array using Large Amounts of Massive Arrays (LAMA) functionality.

Parameters chunksize : int

extra_boundaries : list

adim : list

Returns extra_boundaries, adim : {list, list}

copy(*_omit_special=()*)
Return a deep copy.

Equivalent to `copy.deepcopy(f)`.

Returns

out : The deep copy.

Examples

```
>>> g = f.copy()
```

delattr(*attr*)

Delete a standard or non-standard CF attribute.

Parameters

attr [str] The name of the attribute to delete.

Returns None

Raises

AttributeError : If the variable does not have the named attribute.

Examples

```
>>> f.delattr('standard_name')
>>> f.delattr('foo')
```

dump(*id=None*, *omit=()*)

Return a string containing a full description of the instance.

Parameters

id [str, optional] Set the common prefix of component names. By default the instance's class name is used.

omit [sequence of strs] Omit the given attributes from the description.

Returns

out [str] A string containing the description.

Examples

```
>>> x = v.dump()
>>> print v.dump()
>>> print v.dump(id='variable1')
>>> print v.dump(omit=('long_name',))
```

equals(*other*, *rtol=None*, *atol=None*, *traceback=False*, *ignore=set([])*)

True if two variables are logically equal, False otherwise.

Parameters

other [] The object to compare for equality.

atol [float, optional] The absolute tolerance for all numerical comparisons, By default the value returned by the *ATOL* function is used.

rtol [float, optional] The relative tolerance for all numerical comparisons, By default the value returned by the *RTOL* function is used.

traceback [bool, optional] If True then print a traceback highlighting where the two instances differ.

ignore [set, optional] Omit these attributes from the comparison.

Returns

out [bool] Whether or not the two instances are equal.

Examples

```
>>> f.equals(f)
True
>>> g = f + 1
>>> f.equals(g)
False
>>> g -= 1
```

```
>>> f.equals(g)
True
>>> f.setattr('name', 'name0')
>>> g.setattr('name', 'name1')
>>> f.equals(g)
False
```

expand_dims(*axis*, *dim*, *direction*)
axis is an integer dim is a string direction is a boolean

extract(**args*, ***kwargs*)
Return the instance if it matches the given conditions.

Equivalent to:

```
def extract(f, *args, **kwargs):
    if f.match(*args, **kwargs):
        return f
    raise ValueError('')
```

Parameters

args, kwargs : As for the *match* method.

Returns

out : The variable as an object identity, if it matches the given conditions.

Raises

ValueError : If the variable does not match the conditions.

getattr(*attr*, **default*)
Get a standard or non-standard CF attribute.

Parameters

attr [str] The name of the attribute to get.

default [optional] Return *default* if and only if the variable does not have the named attribute.

Returns

out : The value of the named attribute, or the default value.

Raises

AttributeError : If the variable does not have the named attribute a default value has not been set.

Examples

```
>>> f.getattr('standard_name')
>>> f.getattr('standard_name', None)
>>> f.getattr('foo')
```

hasattr(*attr*)
Return True if the variable has standard or non-standard CF attribute.

Parameters

attr [str] The name of the attribute.

Returns

out [bool] True if the instance has the attribute.

Examples

```
>>> f.hasattr('standard_name')
>>> f.hasattr('foo')
```

match(*attr={}*, *priv={}*)

Determine whether or not a variable matches conditions on its attributes.

Parameters

attr [dict] Dictionary for which each key/value pair is a standard or non-standard CF attribute name and a condition for the attribute to be tested against. If the value is a sequence of conditions then the attribute matches if at least one of the conditions is passed.

In general, a condition is any object and it is passed if the attribute is equal to the object, with the following exceptions:

- If the attribute is string-valued, then the condition may be a regular expression pattern recognised by the *re* module and the condition is passed if the attribute matches the regular expression. Special characters for the start and end of the string are assumed and need not be included. For example, '.*wind' is equivalent to '^.*wind$'.
- For the 'Units' attribute, the condition is passed if the attribute is equivalent (rather than equal) to the object. (Note that this behaviour does not apply to the 'units' attribute.)

priv [dict, optional] Dictionary for which each key/value pair is an attribute name other than their standard and non-standard CF attribute and a condition for the attribute to be tested against. If the value is a sequence of conditions then the attribute matches if at least one of the conditions is passed.

In general, a condition behaves as for *attr*, with the following exception:

- For the *Units* attribute, the condition is passed if the attribute is equivalent (rather than equal) to the object. (Note that this behaviour does not apply to the *units* attribute.)

Returns

out [bool] Whether or not the variable matches the given criteria.

Examples

name(*long_name=False*, *ncvar=False*, *default=None*)

Return a name for the variable.

Returns the standard_name, long_name (if requested) or netCDF variable name (if requested), whichever it finds first, otherwise returns a default name.

Parameters long_name : bool, optional

If True, return the long_name if standard_name does not exist.

ncvar : bool, optional

If True, return ncvar if neither the standard_name nor long_name has already been returned.

default : str, optional

Return default if neither standard_name, long_name nor ncvar has already been returned.

Returns

name [str] The name of the variable.

override_units(*new_units*)

Override the data array units in place.

Not to be confused with setting the *Units* attribute to units which are equivalent to the original units.

This is different to setting the *Units* attribute, as the new units need not be equivalent to the original ones and the data array elements will not be changed to reflect the new units.

Parameters

new_units [str or Units] The new units for the data array.

Returns None

Examples

```
>>> f.Units
<CF Units: hPa>
>>> f.first_datum
100000.0
>>> f.override_units('km')
>>> f.Units
<CF Units: km>
>>> f.first_datum
100000.0
>>> f.override_units(cf.Units('watts'))
>>> f.Units
<CF Units: watts>
>>> f.first_datum
100000.0
```

reverse_dims(*axes=None*)

Reverse the directions of the data array dimensions in place.

Equivalent to indexing the specified dimensions with `::-1`.

Parameters

axes [int or sequence of ints] By default all dimensions are reversed.

Returns

out [list of ints] The axes which were reversed, in arbitrary order.

Examples

```
>>> f.reverse_dims()
>>> f.reverse_dims(1)
>>> f.reverse_dims([0, 1])
```

setattr(*attr*, *value*)

Set a standard or non-standard CF attribute.

Parameters

attr [str] The name of the attribute to set.

value : The value for the attribute.

Returns None

Examples

```
>>> f.setattr('standard_name', 'time')
>>> f.setattr('foo', 12.5)
```

squeeze(*axes=None*)

axes are integers

transpose(*axes=None*)

axes are integers

Data

The *Data* object containing the data array.

Examples

```
>>> f.Data
<CF Data: >
```

Units

The *Units* object containing the units of the data array.

Stores the units and calendar CF attributes in an internally consistent manner. These attributes are mirrored by the *units* and *calendar* attributes respectively.

Examples

```
>>> f.Units
<CF Units: K>
```

add_offset

The add_offset CF attribute.

This attribute is only used when writing to a file on disk.

Examples

```
>>> f.add_offset = -4.0
>>> f.add_offset
-4.0
>>> del f.add_offset
```

array

A numpy array deep copy of the data array.

Changing the returned numpy array does not change the data array.

Examples

```
>>> a = f.array
>>> type(a)
<type 'numpy.ndarray'>
>>> a
array([0, 1, 2, 3, 4])
>>> a[0] = 999
>>> f.array
array([0, 1, 2, 3, 4])
```

attributes

A dictionary of the standard and non-standard CF attributes.

Note that modifying the returned dictionary will not change the attributes.

Examples

```
>>> f.attributes
{'Conventions': 'CF-1.0',
 '_FillValue': 1e+20,
 'cell_methods': <CF CellMethods: time: mean>,
 'experiment_id': 'stabilization experiment (SRES A1B)',
 'long_name': 'Surface Air Temperature',
 'standard_name': 'AIR_TEMP',
 'title': 'SRES A1B',
 'units': 'K'}
```

calendar

The calendar CF attribute.

This attribute is a mirror of the calendar stored in the *Units* attribute.

Examples

```
>>> f.calendar = 'noleap'
>>> f.calendar
'noleap'
>>> del f.calendar
```

comment

The comment CF attribute.

Examples

```
>>> f.comment = 'a comment'
>>> f.comment
'a comment'
>>> del f.comment
```

dtype

Numpy data-type of the data array.

Examples

```
>>> f.dtype
dtype('float64')
```

first_datum

The first element of the data array.

Equivalent to `f.subset[(0,)*f.ndim].array.item()`

Examples

```
>>> f.array
array([[1, 2],
       [3, 4]])
>>> f.first_datum
1
```

history

The history CF attribute.

Examples

```
>>> f.history = 'created on 2012/10/01'
>>> f.history
'created on 2012/10/01'
>>> del f.history
```

is_scalar

True if and only if the data array is a scalar array.

Examples

```
>>> f.array
array(2)
>>> f.is_scalar
True

>>> f.array
array([2])
>>> f.is_scalar
False
```

last_datum

The last element of the data array.

Equivalent to `f.subset[(-1,)*f.ndim].array.item()`

Examples

```
>>> f.array
array([[1, 2],
       [3, 4]])
>>> f.first_datum
4
```

leap_month

The leap_month CF attribute.

Examples

```
>>> f.leap_month = 2
>>> f.leap_month
2
>>> del f.leap_month
```

leap_year

The leap_year CF attribute.

Examples

```
>>> f.leap_year = 1984
>>> f.leap_year
1984
>>> del f.leap_year
```

long_name

The long_name CF attribute.

Examples

```
>>> f.long_name = 'zonal_wind'
>>> f.long_name
'zonal_wind'
>>> del f.long_name
```

mask

The boolean missing data mask of the data array.

Returned as a *Data* object. The mask may be set to 'no missing data' by deleting the attribute.

Examples

```
>>> f.shape
(12, 73, 96)
>>> m = f.mask
>>> type(m)
<cf.data.Data>
>>> m.dtype
dtype('bool')
>>> m.shape
[12, 73, 96]
>>> m.array.shape
(12, 73, 96)

>>> del f.mask
>>> f.array.mask
False
>>> import numpy
>>> f.array.mask is numpy.ma.nomask
True
```

missing_value

The missing_value CF attribute.

This attribute is forced to be consistent with the *_FillValue* attribute as follows:

- Assigning a value to the *missing_value* attribute also assigns the same value to the *_FillValue* attribute whether the latter has been previously set or not.
- Assigning a value to the *_FillValue* attribute also assigns the same value to the *missing_value* attribute, but only if the latter has previously been defined.

Examples

```
>>> f.missing_value = 1e30
>>> f.missing_value
1e30
>>> f._FillValue
1e30
>>> del f.missing_value
>>> f._FillValue
1e30
```

month_lengths
The month_lengths CF attribute.

ndim
Number of dimensions in the data array.

Examples

```
>>> f.shape
(73, 96)
>>> f.ndim
2
```

scale_factor
The scale_factor CF attribute.

This attribute is only used when writing to a file on disk.

Examples

```
>>> f.scale_factor = 10.0
>>> f.scale_factor
10.0
>>> del f.scale_factor
```

shape
Tuple of the data array's dimension sizes.

Examples

```
>>> f.shape
(73, 96)
```

size
Number of elements in the data array.

Examples

```
>>> f.shape
(73, 96)
>>> f.size
7008
```

standard_name
The standard_name CF attribute.

Examples

```
>>> f.standard_name = 'time'
>>> f.standard_name
'time'
>>> del f.standard_name
```

subset

Return a new variable whose data is subsetted.

This attribute may be indexed to select a subset from dimension index values.

Subsetting by indexing

Subsetting by dimension indices uses an extended Python slicing syntax, which is similar numpy array indexing. There are two extensions to the numpy indexing functionality:

- Size 1 dimensions are never removed.

  An integer index i takes the i-th element but does not reduce the rank of the output array by one.

- When advanced indexing is used on more than one dimension, the advanced indices work independently.

  When more than one dimension's slice is a 1-d boolean array or 1-d sequence of integers, then these indices work independently along each dimension (similar to the way vector subscripts work in Fortran), rather than by their elements.

Examples

units

The units CF attribute.

This attribute is a mirror of the units stored in the *Units* attribute.

Examples

```
>>> f.units = 'K'
>>> f.units
'K'
>>> del f.units
```

valid_max

The valid_max CF attribute.

Examples

```
>>> f.valid_max = 100.0
>>> f.valid_max
100.0
>>> del f.valid_max
```

valid_min

The valid_min CF attribute.

Examples

```
>>> f.valid_min = 100.0
>>> f.valid_min
100.0
>>> del f.valid_min
```

valid_range

The valid_range CF attribute.

May be set as a numpy array, a list or a tuple. Always returned as a tuple.

Examples

```
>>> f.valid_range = array([100., 400.])
>>> f.valid_range
(100.0, 400.0)
>>> del f.valid_range
>>> f.valid_range = (50., 450.)
```

varray

A numpy array view of the data array.

Changing the elements of the returned view changes the data array.

Examples

```
>>> a = f.varray
>>> type(a)
<type 'numpy.ndarray'>
>>> a
array([0, 1, 2, 3, 4])
>>> a[0] = 999
>>> f.varray
array([999, 1, 2, 3, 4])
```

8.5.4 cf.VariableList

class cf.**VariableList**(*sequence=()*)

Bases: cf.utils.CfList

An ordered sequence of variables stored in a list-like object.

In some contexts, whether an object is a variable or a variable list is not known and does not matter. So to avoid ungainly type testing, some aspects of the VariableList interface are shared by a variable and vice versa.

Any attribute or method belonging to a variable may be used on a variable list and will be applied independently to each element.

Just as it is straight forward to iterate over the variables in a variable list, a variable will behave like a single element variable list in iterative and indexing contexts.

Initialization

Parameters

sequence [iterable, optional] Define a new list with these elements.

copy()

Return a deep copy.

Equivalent to copy.deepcopy(s).

Returns

out [] The deep copy.

Examples

```
>>> s.copy()
```

count(*value*)

Return the number of occurrences of a given value.

Parameters

value : The value to count.

Returns

out [int] The number of occurrences of *value*.

Examples

```
>>> s
[1, 2, 3, 2, 4, 2]
>>> s.count(1)
1
>>> s.count(2)
3
```

delattr(*attr*)
Delete a public attribute from each element of the list of variables.

dump(**arg*, ***kwargs*)
Return a string containing the full descriptions of each variable in the list.

equals(*other*, *rtol=None*, *atol=None*, *traceback=False*)
True if two instances are logically equal, False otherwise.

Parameters

other [] The object to compare for equality.

atol [float, optional] The absolute tolerance for all numerical comparisons, By default the value returned by the *ATOL* function is used.

rtol [float, optional] The relative tolerance for all numerical comparisons, By default the value returned by the *RTOL* function is used.

traceback [bool, optional] If True then print a traceback highlighting where the two instances differ.

Returns

out [bool] Whether or not the two instances are equal.

extract(**args*, ***kwargs*)
Return the elements which match the given conditions.

The match conditions are passed to each element's *match* method in turn.

Parameters

args, kwargs : As for the variable's *match* method.

Returns

out : A new list containing the matching elements.

Examples

```
>>> f
[<>
 <>]
>>> f.extract(attr={'standard_name': 'air_temperature'})
[<>]
```

getattr(**args*)
Return a built-in list of the public attributes of each element of the list of variables.

hasattr(**args*)
Return a built-in list of the public attributes of each element of the list of variables.

index(*value*, *start=0*, *stop=None*)
Return the first index of a given value.

Parameters value :

start : int, optional

stop : int, optional

Returns out : int

Raises

ValueError : If the given value is not in the list.

Examples

```
>>> s
[1, 2, 3, 2, 4, 2]
>>> s.index(1)
1
>>> s.index(2, start=2)
3
>>> s.index(2, start=2, stop=4)
3
```

insert (*index*, *object*)

Insert an object before the given index in place.

Parameters index : int

object :

Returns None

Examples

```
>>> s
[1, 2, 3]
>>> s.insert(1, 'A')
>>> s
[1, 'A', 2, 3]
```

match (**args*, ***kwargs*)

Return a list of booleans showing which elements match the given conditions.

The match conditions are passed to each element's *match* method in turn.

Parameters

args, kwargs : As for the variable's *match* method.

Returns

out [list] A built-in list of booleans showing which elements match the conditions.

Examples

```
>>> f
[<>
 <>]
>>> f.match(attr={'standard_name': 'air_temperature'})
[True, False]
```

name (**arg*, ***kwargs*)

Return a built-in list of the names of each element of the list of variables.

setattr (*attr*, *value*)

Set a public attribute from each element of the list of variables.

subset

Subset each variable in the list, returning a new list of variables.

Examples

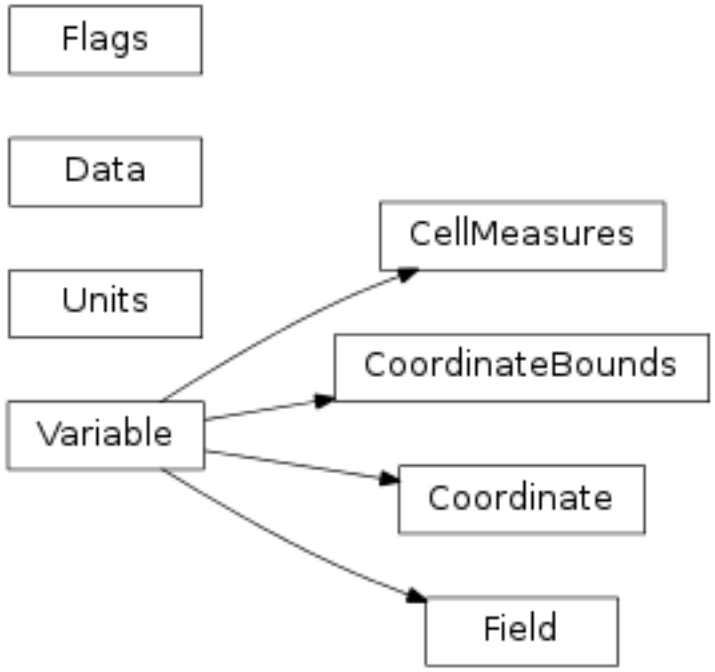
```
>>> vl
[<CF Variable: air_temperature(73, 96)>,
 <CF Variable: air_temperature(73, 96)>]
>>> vl.subset[0,0]
```

```
[<CF Variable: air_temperature(1,1)>,
 <CF Variable: air_temperature(1,1)>]
```

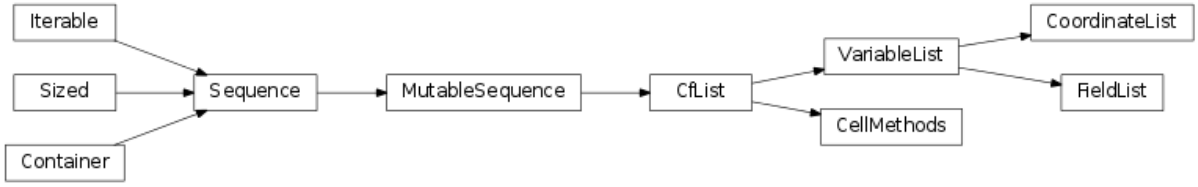
8.6 Inheritance diagrams

The classes defined by the cf package inherit as follows.

Classes `Data`, `Units`, `Flags` and `Variable` all inherit directly from the built-in `object`[2]. Classes `Field`, `Coordinate`, `CoordinateBounds` and `CellMeasures` are all sub-classes of `Variable`:



Classes `CfList`, `VariableList`, `FieldList`, `CoordinateList` and `CellMethods` are list-like objects and inherit the built-in `MutableSequence`[3] abstract base class:



This inheritance diagram shows, for example, that a `FieldList` object is an iterable [4] which supports efficient element access using integer indices and defines a `len()` method that returns the length of the sequence.

Classes `CfDict`, `Space` and `Transform` are dictionary-like objects and inherit the built-in `MutableMapping`[5] abstract base class.

[2] http://docs.python.org/2.7/library/functions.html#object
[3] http://docs.python.org/2.7/library/collections.html#collections.MutableSequence
[4] An object capable of returning its members one at a time.
[5] http://docs.python.org/2.7/library/collections.html#collections.MutableMapping

This inheritance diagram shows, for example, that a `Space` object supports arbitrary, hashable key lookups and implements dictionary-like methods, such as `pop()`.

CHAPTER

NINE

CONSTANTS

Useful constants are stored in the CONSTANTS dictionary.

9.1 cf.CONSTANTS

cf.**CONSTANTS**

A dictionary of useful constants.

Whilst the dictionary may be modified directly, it is safer to retrieve and set the values with a function where one is provided. This is due to interdependencies between some values.

Keys

ATOL [float] The value of absolute tolerance for testing numerically tolerant equality. Retrieved and set with ATOL.

CHUNKSIZE [int] The chunk size (in bytes) for data storage and processing. Retrieved and set with CHUNKSIZE.

FM_THRESHOLD [float] The minimum amount of memory (in kibibytes) to be kept free as a temporary work space. Retrieved and set with FM_THRESHOLD.

MINNCFM [int] The number of chunk sizes to be kept free a temporary work space.

RTOL [float] The value of relative tolerance for testing numerically tolerant equality. Retrieved and set with RTOL.

TEMPDIR [str] The location to store temporary files. By default the directory returned by the tempfile.gettempdir[1] function is used.

Examples

```
>>> cf.CONSTANTS
{'ATOL': 2.2204460492503131e-16,
 'CHUNKSIZE': 104857600,
 'FM_THRESHOLD': 1024000.0,
 'MINNCFM': 10,
 'RTOL': 2.2204460492503131e-16,
 'TEMPDIR': '/tmp'}
>>> cf.CONSTANTS['MINNCFM'] = 15
>>> cf.CONSTANTS['TEMPDIR'] = '/home/me/tmp'
>>> cf.CHUNKSIZE(2**30)
>>> cf.CONSTANTS['CHUNKSIZE']
1073741824
>>> cf.CONSTANTS['FM_THRESHOLD']
10485760.0
```

[1] http://docs.python.org/2.7/library/tempfile.html#tempfile.gettempdir

Part IV

Indices and tables

- *genindex*
- *modindex*
- *search*

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W