Package 'CFtime'

November 12, 2025

Title Using CF-Compliant Calendars with Climate Projection Data

Version 1.7.2

Description Support for all calendars as specified in the Climate and Forecast (CF) Metadata Conventions for climate and forecasting data. The CF Metadata Conventions is widely used for distributing files with climate observations or projections, including the Coupled Model Intercomparison Project (CMIP) data used by climate change scientists and the Intergovernmental Panel on Climate Change (IPCC). This package specifically allows the user to work with any of the CF-compliant calendars (many of which are not compliant with POSIXt). The CF time coordinate is formally defined in the CF Metadata Conventions document available at <https://cfconventions.org/Data/cf-conventions/cf-conventions-1.12/cf-conventions.html#time-coordinate>.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.3

Imports R6

Suggests rmarkdown, testthat (>= 3.0.0), stringr

URL https://r-cf.github.io/CFtime/, https://github.com/R-CF/CFtime

BugReports https://github.com/R-CF/CFtime/issues

Config/testthat/edition 3

Collate 'api.R' 'CFCalendar.R' 'CFCalendar360.R' 'CFCalendar365.R' 'CFCalendar366.R' 'CFCalendarJulian.R' 'CFCalendarNone.R' 'CFCalendarProleptic.R' 'CFCalendarStandard.R' 'CFCalendarTAI.R' 'CFCalendarUTC.R' 'CFClimatology.R' 'CFtime-package.R' 'CFtime.R' 'helpers.R' 'zzz.R'

Config/Needs/website rmarkdown

NeedsCompilation no

Author Patrick Van Laake [aut, cre, cph]

Maintainer Patrick Van Laake <patrick@vanlaake.net>

Repository CRAN

Date/Publication 2025-11-12 15:40:12 UTC

Contents

+.CFTime   *Extend a CFTime object*

Description

A CFTime instance can be extended with this operator, using values from another CFTime instance, or a vector of numeric offsets or character timestamps. If the values come from another CFTime instance, the calendars of the two instances must be compatible If the calendars of the CFTime instances are not compatible, an error is thrown.

Usage

```
## S3 method for class 'CFTime'
e1 + e2
```

Arguments

e1	Instance of the CFTime class.
e2	Instance of the CFTime class with a calendar compatible with that of argument e1, or a numeric vector with offsets from the origin of argument e1, or a vector of character timestamps in ISO8601 or UDUNITS format.

Details

The resulting CFTime instance will have the offsets of the original CFTime instance, appended with offsets from argument e2 in the order that they are specified. If the new sequence of offsets is not monotonically increasing a warning is generated (the COARDS metadata convention requires offsets to be monotonically increasing).

There is no reordering or removal of duplicates. This is because the time series are usually associated with a data set and the correspondence between the data in the files and the CFTime instance is thus preserved. When merging the data sets described by this time series, the order must be identical to the merging here.

Note that when adding multiple vectors of offsets to a CFTime instance, it is more efficient to first concatenate the vectors and then do a final addition to the CFTime instance. So avoid CFtime(definition, calendar, e1) + CFtime(definition, calendar, e2) + CFtime(definition, calendar, e3) + ... but rather do CFtime(definition, calendar) + c(e1, e2, e3, ...). It is the responsibility of the operator to ensure that the offsets of the different data sets are in reference to the same calendar.

Note also that RNetCDF and ncdf4 packages both return the values of the "time" dimension as a 1-dimensional array. You have to dim(time_values) <- NULL to de-class the array to a vector before adding offsets to an existing CFtime instance.

Any bounds that were set will be removed. Use bounds() to retrieve the bounds of the individual CFTime instances and then set them again after merging the two instances.

Value

A CFTime object with the offsets of argument e1 extended by the values from argument e2.

Examples

```
e1 <- CFtime("days since 1850-01-01", "gregorian", 0:364)
e2 <- CFtime("days since 1850-01-01 00:00:00", "standard", 365:729)
e1 + e2
```

==.CFTime   *Equivalence of CFTime objects*

Description

This operator can be used to test if two [CFTime](#) objects represent the same CF-convention time coordinates. Two CFTime objects are considered equivalent if they have an equivalent calendar and the same offsets.

Usage

```
## S3 method for class 'CFTime'
e1 == e2
```

Arguments

e1, e2   Instances of the CFTime class.

Value

TRUE if the CFTime objects are equivalent, FALSE otherwise.

Examples

```
e1 <- CFtime("days since 1850-01-01", "gregorian", 0:364)
e2 <- CFtime("days since 1850-01-01 00:00:00", "standard", 0:364)
e1 == e2
```

as.character.CFTime   *Return the timestamps contained in the* CFTime *instance*

Description

Return the timestamps contained in the CFTime instance

Usage

```
## S3 method for class 'CFTime'
as.character(x, ...)
```

Arguments

x   The CFTime instance whose timestamps will be returned.

...   Ignored.

Value

The timestamps in the specified `CFTime` instance.

Examples

```
t <- CFtime("days since 1850-01-01", "julian", 0:364)
as.character(t)
```

---

as_timestamp   *Create a vector that represents CF timestamps*

Description

This function generates a vector of character strings or `POSIXct`s that represent the date and time in a selectable combination for each offset.

Usage

```
as_timestamp(t, format = NULL, asPOSIX = FALSE)
```

Arguments

t	The `CFTime` instance that contains the offsets to use.
format	character. A character string with either of the values "date" or "timestamp". If the argument is not specified, the format used is "timestamp" if there is time information, "date" otherwise.
asPOSIX	logical. If `TRUE`, for "standard", "gregorian" and "proleptic_gregorian" calendars the output is a vector of `POSIXct` - for other calendars an error will be thrown. Default value is `FALSE`.

Details

The character strings use the format `YYYY-MM-DDThh:mm:ss±hhmm`, depending on the `format` specifier. The date in the string is not necessarily compatible with `POSIXt` - in the `360_day` calendar `2017-02-30` is valid and `2017-03-31` is not.

For the "proleptic_gregorian" calendar the output can also be generated as a vector of `POSIXct` values by specifying `asPOSIX = TRUE`. The same is possible for the "standard" and "gregorian" calendars but only if all timestamps fall on or after 1582-10-15.

Value

A character vector where each element represents a moment in time according to the `format` specifier.

See Also

The [CFTime](#) `format()` method gives greater flexibility through the use of `strptime`-like format specifiers.

Examples

```
t <- CFtime("hours since 2020-01-01", "standard", seq(0, 24, by = 0.25))
as_timestamp(t, "timestamp")

t2 <- CFtime("days since 2002-01-21", "standard", 0:20)
tail(as_timestamp(t2, asPOSIX = TRUE))

tail(as_timestamp(t2))

tail(as_timestamp(t2 + 1.5))
```

---

bounds   *Bounds of the time offsets*

Description

CF-compliant netCDF files store time information as a single offset value for each step along the dimension, typically centered on the valid interval of the data (e.g. 12-noon for day data). Optionally, the lower and upper values of the valid interval are stored in a so-called "bounds" variable, as an array with two rows (lower and higher value) and a column for each offset. With function bounds()<- those bounds can be set for a CFTime instance. The bounds can be retrieved with the bounds() function.

Usage

```
bounds(x, format)

bounds(x) <- value
```

Arguments

x	A CFTime instance.
format	Optional. A single string with format specifiers, see format() for details.
value	A matrix (or array) with dimensions (2, length(offsets)) giving the lower (first row) and higher (second row) bounds of each offset (this is the format that the CF Metadata Conventions uses for storage in netCDF files). Use NULL to unset any previously set bounds.

Value

If bounds have been set, an array of bounds values with dimensions (2, length(offsets)). The first row gives the lower bound, the second row the upper bound, with each column representing an offset of x. If the format argument is specified, the bounds values are returned as strings according to the format. NULL when no bounds have been set.

Examples

```
t <- CFtime("days since 2024-01-01", "standard", seq(0.5, by = 1, length.out = 366))
as_timestamp(t)[1:3]
bounds(t) <- rbind(0:365, 1:366)
bounds(t)[, 1:3]
bounds(t, "%d-%b-%Y")[, 1:3]
```

CFCalendar   *Basic CF calendar*

Description

This class represents a basic CF calendar. It should not be instantiated directly; instead, use one of the descendant classes.

This internal class stores the information to represent date and time values using the CF conventions. An instance is created by the exported CFTime class, which also exposes the relevant properties of this class.

The following calendars are supported:

- `gregorian\standard`, the international standard calendar for civil use.
- `proleptic_gregorian`, the standard calendar but extending before 1582-10-15 when the Gregorian calendar was adopted.
- `tai`, International Atomic Time clock with dates expressed using the Gregorian calendar.
- `utc`, Coordinated Universal Time clock with dates expressed using the Gregorian calendar.
- `julian`, every fourth year is a leap year.
- `noleap\365_day`, all years have 365 days.
- `all_leap\366_day`, all years have 366 days.
- `360_day`, all years have 360 days, divided over 12 months of 30 days.

Active bindings

name (read-only) Name of the calendar, as per the CF Metadata Conventions.

definition (read-only) The string that defines the units and the origin, as per the CF Metadata Conventions.

unit (read-only) The numeric id of the unit of the calendar.

prefix_id (read-only) The index value of the prefix of the time unit. If the unit does not have a prefix, returns the value 0L.

origin (read-only) data.frame with fields for the origin of the calendar.

origin_date (read-only) Character string with the date of the calendar.

origin_time (read-only) Character string with the time of the calendar.

timezone (read-only) Character string with the time zone of the origin of the calendar.

Methods

Public methods:

- CFCalendar$new()
- CFCalendar$print()
- CFCalendar$valid_days()
- CFCalendar$add_day()
- CFCalendar$POSIX_compatible()
- CFCalendar$is_compatible()
- CFCalendar$is_equivalent()
- CFCalendar$parse()
- CFCalendar$offsets2time()
- CFCalendar$clone()

Method new(): Create a new CF calendar.

Usage:

CFCalendar$new(nm, definition)

Arguments:

nm The name of the calendar. This must follow the CF Metadata Conventions.

definition The string that defines the units and the origin, as per the CF Metadata Conventions.

Method print(): Print information about the calendar to the console.

Usage:

CFCalendar$print(...)

Arguments:

... Ignored.

Returns: self, invisibly.

Method valid_days(): Indicate which of the supplied dates are valid.

Usage:

CFCalendar$valid_days(ymd)

Arguments:

ymd data.frame with dates parsed into their parts in columns year, month and day. Any other columns are disregarded.

Returns: NULL. A warning will be generated to the effect that a descendant class should be used for this method.

Method add_day(): Add a day to the supplied dates.

Usage:

CFCalendar$add_day(ymd)

Arguments:

ymd data.frame with dates parsed into their parts in columns year, month and day. Any other columns are disregarded.

Returns: A data.frame like argument ymd but with a day added for every row.

Method POSIX_compatible(): Indicate if the time series described using this calendar can be safely converted to a standard date-time type (POSIXct, POSIXlt, Date).

Only the 'standard' calendar and the 'proleptic_gregorian' calendar when all dates in the time series are more recent than 1582-10-15 (inclusive) can be safely converted, so this method returns FALSE by default to cover the majority of cases.

Usage:
CFCalendar$POSIX_compatible(offsets)

Arguments:

offsets The offsets from the CFtime instance.

Returns: FALSE by default.

Method is_compatible(): This method tests if the CFCalendar instance in argument cal is compatible with self, meaning that they are of the same class and have the same unit and prefix. Calendars "standard", and "gregorian" are compatible, as are the pairs of "365_day" and "no_leap", and "366_day" and "all_leap".

Usage:
CFCalendar$is_compatible(cal)

Arguments:

cal Instance of a descendant of the CFCalendar class.

Returns: TRUE if the instance in argument cal is compatible with self, FALSE otherwise.

Method is_equivalent(): This method tests if the CFCalendar instance in argument cal is equivalent to self, meaning that they are of the same class, have the same unit and prefix, and equivalent origins. Calendars "standard", and "gregorian" are equivalent, as are the pairs of "365_day" and "no_leap", and "366_day" and "all_leap".

Note that the origins need not be identical, but their parsed values have to be. "2000-01" is parsed the same as "2000-01-01 00:00:00", for instance.

Usage:
CFCalendar$is_equivalent(cal)

Arguments:

cal Instance of a descendant of the CFCalendar class.

Returns: TRUE if the instance in argument cal is equivalent to self, FALSE otherwise.

Method parse(): Parsing a vector of date-time character strings into parts.

Usage:
CFCalendar$parse(d)

Arguments:

d character. A character vector of date-times.

Returns: A data.frame with columns year, month, day, hour, minute, second, time zone, and offset. Invalid input data will appear as NA.

Method offsets2time(): Decompose a vector of offsets, in units of the calendar, to their timestamp values. This adds a specified amount of time to the origin of a CFTime object.

This method may introduce inaccuracies where the calendar unit is "months" or "years", due to the ambiguous definition of these units.

Usage:

CFCalendar$offsets2time(offsets = NULL)

Arguments:

offsets Vector of numeric offsets to add to the origin of the calendar.

Returns: A data.frame with columns for the timestamp elements and as many rows as there are offsets.

Method clone(): The objects of this class are cloneable with this method.

Usage:

CFCalendar$clone(deep = FALSE)

Arguments:

deep Whether to make a deep clone.

References

https://cfconventions.org/Data/cf-conventions/cf-conventions-1.12/cf-conventions.html#calendar

---

CFCalendar360   *360-day CF calendar*

Description

This class represents a CF calendar of 360 days per year, evenly divided over 12 months of 30 days. This calendar is obviously not compatible with the standard POSIXt calendar.

This calendar supports dates before year 1 and includes the year 0.

Super class

[CFtime::CFCalendar](#) -> CFCalendar360

Methods

Public methods:

- [CFCalendar360$new()](#)
- [CFCalendar360$valid_days()](#)
- [CFCalendar360$month_days()](#)
- [CFCalendar360$leap_year()](#)

- `CFCalendar360$date2offset()`
- `CFCalendar360$offset2date()`
- `CFCalendar360$clone()`

Method new(): Create a new CF calendar.

Usage:
`CFCalendar360$new(nm, definition)`

Arguments:

nm The name of the calendar. This must be "360_day". This argument is superfluous but maintained to be consistent with the initialization methods of the parent and sibling classes.

definition The string that defines the units and the origin, as per the CF Metadata Conventions.

Returns: A new instance of this class.

Method valid_days(): Indicate which of the supplied dates are valid.

Usage:
`CFCalendar360$valid_days(ymd)`

Arguments:

ymd data.frame with dates parsed into their parts in columns year, month and day. Any other columns are disregarded.

Returns: Logical vector with the same length as argument ymd has rows with TRUE for valid days and FALSE for invalid days, or NA where the row in argument ymd has NA values.

Method month_days(): Determine the number of days in the month of the calendar.

Usage:
`CFCalendar360$month_days(ymd = NULL)`

Arguments:

ymd data.frame with dates parsed into their parts in columns year, month and day. Any other columns are disregarded.

Returns: A vector indicating the number of days in each month for the dates supplied as argument ymd. If no dates are supplied, the number of days per month for the calendar as a vector of length 12.

Method leap_year(): Indicate which years are leap years.

Usage:
`CFCalendar360$leap_year(yr)`

Arguments:

yr Integer vector of years to test.

Returns: Logical vector with the same length as argument yr. Since this calendar does not use leap days, all values will be FALSE, or NA where argument yr is NA.

Method date2offset(): Calculate difference in days between a data.frame of time parts and the origin.

Usage:

CFCalendar360$date2offset(x)

Arguments:

x data.frame. Dates to calculate the difference for.

Returns: Integer vector of a length equal to the number of rows in argument x indicating the number of days between x and the origin, or NA for rows in x with NA values.

Method offset2date(): Calculate date parts from day differences from the origin. This only deals with days as these are impacted by the calendar. Hour-minute-second timestamp parts are handled in CFCalendar.

Usage:

CFCalendar360$offset2date(x)

Arguments:

x Integer vector of days to add to the origin.

Returns: A data.frame with columns 'year', 'month' and 'day' and as many rows as the length of vector x.

Method clone(): The objects of this class are cloneable with this method.

Usage:

CFCalendar360$clone(deep = FALSE)

Arguments:

deep Whether to make a deep clone.

CFCalendar365 *365-day CF calendar*

Description

This class represents a CF calendar of 365 days per year, having no leap days in any year. This calendar is not compatible with the standard POSIXt calendar.

This calendar supports dates before year 1 and includes the year 0.

Super class

CFtime::CFCalendar -> CFCalendar365

Methods

Public methods:

- CFCalendar365$new()
- CFCalendar365$valid_days()
- CFCalendar365$month_days()
- CFCalendar365$leap_year()

- `CFCalendar365$date2offset()`
- `CFCalendar365$offset2date()`
- `CFCalendar365$clone()`

Method new(): Create a new CF calendar of 365 days per year.

Usage:
`CFCalendar365$new(nm, definition)`

Arguments:

nm The name of the calendar. This must be "365_day" or "noleap".

definition The string that defines the units and the origin, as per the CF Metadata Conventions.

Returns: A new instance of this class.

Method valid_days(): Indicate which of the supplied dates are valid.

Usage:
`CFCalendar365$valid_days(ymd)`

Arguments:

ymd data.frame with dates parsed into their parts in columns year, month and day. Any other columns are disregarded.

Returns: Logical vector with the same length as argument ymd has rows with TRUE for valid days and FALSE for invalid days, or NA where the row in argument ymd has NA values.

Method month_days(): Determine the number of days in the month of the calendar.

Usage:
`CFCalendar365$month_days(ymd = NULL)`

Arguments:

ymd data.frame, optional, with dates parsed into their parts.

Returns: A vector indicating the number of days in each month for the dates supplied as argument ymd. If no dates are supplied, the number of days per month for the calendar as a vector of length 12.

Method leap_year(): Indicate which years are leap years.

Usage:
`CFCalendar365$leap_year(yr)`

Arguments:

yr Integer vector of years to test.

Returns: Logical vector with the same length as argument yr. Since this calendar does not use leap days, all values will be FALSE, or NA where argument yr is NA.

Method date2offset(): Calculate difference in days between a data.frame of time parts and the origin.

Usage:
`CFCalendar365$date2offset(x)`

Arguments:

x data.frame. Dates to calculate the difference for.

Returns: Integer vector of a length equal to the number of rows in argument x indicating the number of days between x and the origin, or NA for rows in x with NA values.

Method offset2date(): Calculate date parts from day differences from the origin. This only deals with days as these are impacted by the calendar. Hour-minute-second timestamp parts are handled in CFCalendar.

Usage:
CFCalendar365$offset2date(x)

Arguments:

x Integer vector of days to add to the origin.

Returns: A data.frame with columns 'year', 'month' and 'day' and as many rows as the length of vector x.

Method clone(): The objects of this class are cloneable with this method.

Usage:
CFCalendar365$clone(deep = FALSE)

Arguments:

deep Whether to make a deep clone.

CFCalendar366 *366-day CF calendar*

Description

This class represents a CF calendar of 366 days per year, having leap days in every year. This calendar is not compatible with the standard POSIXt calendar.

This calendar supports dates before year 1 and includes the year 0.

Super class

CFtime::CFCalendar -> CFCalendar366

Methods

Public methods:

- CFCalendar366$new()
- CFCalendar366$valid_days()
- CFCalendar366$month_days()
- CFCalendar366$leap_year()
- CFCalendar366$date2offset()
- CFCalendar366$offset2date()

- CFCalendar366$clone()

Method new(): Create a new CF calendar of 366 days per year.

Usage:

CFCalendar366$new(nm, definition)

Arguments:

nm The name of the calendar. This must be "366_day" or "all_leap".

definition The string that defines the units and the origin, as per the CF Metadata Conventions.

Returns: A new instance of this class.

Method valid_days(): Indicate which of the supplied dates are valid.

Usage:

CFCalendar366$valid_days(ymd)

Arguments:

ymd data.frame with dates parsed into their parts in columns year, month and day. Any other columns are disregarded.

Returns: Logical vector with the same length as argument ymd has rows with TRUE for valid days and FALSE for invalid days, or NA where the row in argument ymd has NA values.

Method month_days(): Determine the number of days in the month of the calendar.

Usage:

CFCalendar366$month_days(ymd = NULL)

Arguments:

ymd data.frame, optional, with dates parsed into their parts.

Returns: A vector indicating the number of days in each month for the dates supplied as argument ymd. If no dates are supplied, the number of days per month for the calendar as a vector of length 12.

Method leap_year(): Indicate which years are leap years.

Usage:

CFCalendar366$leap_year(yr)

Arguments:

yr Integer vector of years to test.

Returns: Logical vector with the same length as argument yr. Since in this calendar all years have a leap day, all values will be TRUE, or NA where argument yr is NA.

Method date2offset(): Calculate difference in days between a data.frame of time parts and the origin.

Usage:

CFCalendar366$date2offset(x)

Arguments:

x `data.frame`. Dates to calculate the difference for.

Returns: Integer vector of a length equal to the number of rows in argument x indicating the number of days between x and the `origin`, or NA for rows in x with NA values.

Method `offset2date()`: Calculate date parts from day differences from the origin. This only deals with days as these are impacted by the calendar. Hour-minute-second timestamp parts are handled in [CFCalendar](#).

Usage:
```
CFCalendar366$offset2date(x)
```

Arguments:

x Integer vector of days to add to the origin.

Returns: A `data.frame` with columns 'year', 'month' and 'day' and as many rows as the length of vector x.

Method `clone()`: The objects of this class are cloneable with this method.

Usage:
```
CFCalendar366$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

CFCalendarJulian   *Julian CF calendar*

Description

This class represents a Julian calendar of 365 days per year, with every fourth year being a leap year of 366 days. The months and the year align with the standard calendar. This calendar is not compatible with the standard POSIXt calendar.

This calendar starts on 1 January of year 1: 0001-01-01 00:00:00. Any dates before this will generate an error.

Super class

[CFtime::CFCalendar](#) -> CFCalendarJulian

Methods

Public methods:

- [CFCalendarJulian$new()](#)
- [CFCalendarJulian$valid_days()](#)
- [CFCalendarJulian$month_days()](#)
- [CFCalendarJulian$leap_year()](#)
- [CFCalendarJulian$date2offset()](#)

- CFCalendarJulian$offset2date()
- CFCalendarJulian$clone()

Method new(): Create a new CF calendar.

Usage:

CFCalendarJulian$new(nm, definition)

Arguments:

nm The name of the calendar. This must be "julian". This argument is superfluous but maintained to be consistent with the initialization methods of the parent and sibling classes.

definition The string that defines the units and the origin, as per the CF Metadata Conventions.

Returns: A new instance of this class.

Method valid_days(): Indicate which of the supplied dates are valid.

Usage:

CFCalendarJulian$valid_days(ymd)

Arguments:

ymd data.frame with dates parsed into their parts in columns year, month and day. Any other columns are disregarded.

Returns: Logical vector with the same length as argument ymd has rows with TRUE for valid days and FALSE for invalid days, or NA where the row in argument ymd has NA values.

Method month_days(): Determine the number of days in the month of the calendar.

Usage:

CFCalendarJulian$month_days(ymd = NULL)

Arguments:

ymd data.frame, optional, with dates parsed into their parts.

Returns: A vector indicating the number of days in each month for the dates supplied as argument ymd. If no dates are supplied, the number of days per month for the calendar as a vector of length 12, for a regular year without a leap day.

Method leap_year(): Indicate which years are leap years.

Usage:

CFCalendarJulian$leap_year(yr)

Arguments:

yr Integer vector of years to test.

Returns: Logical vector with the same length as argument yr. NA is returned where elements in argument yr are NA.

Method date2offset(): Calculate difference in days between a data.frame of time parts and the origin.

Usage:

CFCalendarJulian$date2offset(x)

Arguments:

x `data.frame`. Dates to calculate the difference for.

Returns: Integer vector of a length equal to the number of rows in argument x indicating the number of days between x and the origin of the calendar, or NA for rows in x with NA values.

Method `offset2date()`: Calculate date parts from day differences from the origin. This only deals with days as these are impacted by the calendar. Hour-minute-second timestamp parts are handled in [CFCalendar](#).

Usage:

```
CFCalendarJulian$offset2date(x)
```

Arguments:

x Integer vector of days to add to the origin.

Returns: A `data.frame` with columns 'year', 'month' and 'day' and as many rows as the length of vector x.

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```
CFCalendarJulian$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

CFCalendarNone   *CF calendar with no annual cycle*

Description

This class represents a CF calendar with no annual cycle. All datetimes in the calendar are the same. This is useful only for repeated experiments simulating a fixed time in the year.

Super class

[CFtime::CFCalendar](#) -> CFCalendarNone

Methods

Public methods:

- [CFCalendarNone$new()](#)
- [CFCalendarNone$valid_days()](#)
- [CFCalendarNone$month_days()](#)
- [CFCalendarNone$leap_year()](#)
- [CFCalendarNone$date2offset()](#)
- [CFCalendarNone$offset2date()](#)
- [CFCalendarNone$clone()](#)

Method new(): Create a new CF calendar.

Usage:

CFCalendarNone$new(nm, definition)

Arguments:

nm The name of the calendar. This must be "none". This argument is superfluous but maintained to be consistent with the initialization methods of the parent and sibling classes.

definition The string that defines the units and the origin, as per the CF Metadata Conventions.

Returns: A new instance of this class.

Method valid_days(): Indicate which of the supplied dates are valid. In this calendar only one date is valid, namely the one that the calendar is initialized with.

Usage:

CFCalendarNone$valid_days(ymd)

Arguments:

ymd data.frame with dates parsed into their parts in columns year, month and day. Any other columns are disregarded.

Returns: Logical vector with the same length as argument ymd has rows with TRUE for valid days and FALSE for invalid days, or NA where the row in argument ymd has NA values.

Method month_days(): Determine the number of days in the month of the calendar. This always returns a vector of NA values because this method has no meaning for this class.

Usage:

CFCalendarNone$month_days(ymd = NULL)

Arguments:

ymd data.frame with dates parsed into their parts in columns year, month and day. Any other columns are disregarded.

Returns: A vector with NA values for the dates supplied as argument ymd. If no dates are supplied, a vector of NA values of length 12.

Method leap_year(): Indicate which years are leap years. This always returns a vector of NA values because this method has no meaning for this class.

Usage:

CFCalendarNone$leap_year(yr)

Arguments:

yr Integer vector of years to test.

Returns: Vector with the same length as argument yr with NA values.

Method date2offset(): Calculate difference in days between a data.frame of time parts and the origin. The difference is always 0, given that this is not a true calendar and there can be no calculations to determine any difference.

Usage:

CFCalendarNone$date2offset(x)

Arguments:

x data.frame. Dates to calculate the difference for.

Returns: Integer vector of a length equal to the number of rows in argument x indicating 0 for the days that equal the origin of the calendar, or NA otherwise.

Method offset2date(): Calculate date parts from day differences from the origin. This always returns 0's for the year, month and day as other values are not valid in this calendar. Hour-minute-second datetime parts are handled in [CFCalendar](#).

Usage:

CFCalendarNone$offset2date(x)

Arguments:

x Integer vector of days to add to the origin.

Returns: A data.frame with columns 'year', 'month' and 'day' and as many rows as the length of vector x. Rows with values of x other than 0 will return the values for the origin of the calendar nonetheless, in accordance with the CF Metadata Conventions.

Method clone(): The objects of this class are cloneable with this method.

Usage:

CFCalendarNone$clone(deep = FALSE)

Arguments:

deep Whether to make a deep clone.

CFCalendarProleptic   *Proleptic Gregorian CF calendar*

Description

This class represents a standard CF calendar, but with the Gregorian calendar extended backwards to before the introduction of the Gregorian calendar. This calendar is compatible with the standard POSIXt calendar, but note that daylight savings time is not considered.

This calendar includes dates 1582-10-14 to 1582-10-05 (the gap between the Gregorian and Julian calendars, which is observed by the standard calendar), and extends to years before the year 1, including year 0.

Super class

[CFtime::CFCalendar](#) -> CFCalendarProleptic

Methods

Public methods:

- CFCalendarProleptic$new()
- CFCalendarProleptic$valid_days()
- CFCalendarProleptic$month_days()
- CFCalendarProleptic$leap_year()
- CFCalendarProleptic$POSIX_compatible()
- CFCalendarProleptic$date2offset()
- CFCalendarProleptic$offset2date()
- CFCalendarProleptic$clone()

Method new(): Create a new CF calendar.

Usage:

CFCalendarProleptic$new(nm, definition)

Arguments:

nm The name of the calendar. This must be "proleptic_gregorian". This argument is superfluous but maintained to be consistent with the initialization methods of the parent and sibling classes.

definition The string that defines the units and the origin, as per the CF Metadata Conventions.

Returns: A new instance of this class.

Method valid_days(): Indicate which of the supplied dates are valid.

Usage:

CFCalendarProleptic$valid_days(ymd)

Arguments:

ymd data.frame with dates parsed into their parts in columns year, month and day. Any other columns are disregarded.

Returns: Logical vector with the same length as argument ymd has rows with TRUE for valid days and FALSE for invalid days, or NA where the row in argument ymd has NA values.

Method month_days(): Determine the number of days in the month of the calendar.

Usage:

CFCalendarProleptic$month_days(ymd = NULL)

Arguments:

ymd data.frame, optional, with dates parsed into their parts.

Returns: Integer vector indicating the number of days in each month for the dates supplied as argument ymd. If no dates are supplied, the number of days per month for the calendar as a vector of length 12, for a regular year without a leap day.

Method leap_year(): Indicate which years are leap years.

Usage:

```
CFCalendarProleptic$leap_year(yr)
```

Arguments:

yr Integer vector of years to test.

Returns: Logical vector with the same length as argument yr. NA is returned where elements in argument yr are NA.

Method POSIX_compatible(): Indicate if the time series described using this calendar can be safely converted to a standard date-time type (POSIXct, POSIXlt, Date).

Usage:
```
CFCalendarProleptic$POSIX_compatible(offsets)
```

Arguments:

offsets The offsets from the CFtime instance.

Returns: TRUE.

Method date2offset(): Calculate difference in days between a data.frame of time parts and the origin.

Usage:
```
CFCalendarProleptic$date2offset(x)
```

Arguments:

x data.frame. Dates to calculate the difference for.

Returns: Integer vector of a length equal to the number of rows in argument x indicating the number of days between x and the origin, or NA for rows in x with NA values.

Method offset2date(): Calculate date parts from day differences from the origin. This only deals with days as these are impacted by the calendar. Hour-minute-second timestamp parts are handled in [CFCalendar](#).

Usage:
```
CFCalendarProleptic$offset2date(x)
```

Arguments:

x Integer vector of days to add to the origin.

Returns: A data.frame with columns 'year', 'month' and 'day' and as many rows as the length of vector x.

Method clone(): The objects of this class are cloneable with this method.

Usage:
```
CFCalendarProleptic$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

CFCalendarStandard   *Standard CF calendar*

Description

This class represents a standard calendar of 365 or 366 days per year. This calendar is compatible with the standard POSIXt calendar for periods after the introduction of the Gregorian calendar, 1582-10-15 00:00:00. The calendar starts at 0001-01-01 00:00:00, e.g. the start of the Common Era.

Note that this calendar, despite its name, is not the same as that used in ISO8601 or many computer systems for periods prior to the introduction of the Gregorian calendar. Use of the "proleptic_gregorian" calendar is recommended for periods before or straddling the introduction date, as that calendar is compatible with POSIXt on most OSes.

Super class

[CFtime::CFCalendar](#) -> CFCalendarStandard

Methods

Public methods:

- [CFCalendarStandard$new()](#)
- [CFCalendarStandard$valid_days()](#)
- [CFCalendarStandard$is_gregorian_date()](#)
- [CFCalendarStandard$POSIX_compatible()](#)
- [CFCalendarStandard$month_days()](#)
- [CFCalendarStandard$leap_year()](#)
- [CFCalendarStandard$date2offset()](#)
- [CFCalendarStandard$offset2date()](#)
- [CFCalendarStandard$clone()](#)

Method new(): Create a new CF calendar. When called with the deprecated 'gregorian' it is automatically converted to the equivalent 'standard'.

Usage:

CFCalendarStandard$new(nm, definition)

Arguments:

nm The name of the calendar. This must be "standard" or "gregorian" (deprecated).

definition The string that defines the units and the origin, as per the CF Metadata Conventions.

Returns: A new instance of this class.

Method valid_days(): Indicate which of the supplied dates are valid.

Usage:

```
CFCalendarStandard$valid_days(ymd)
```

Arguments:

ymd data.frame with dates parsed into their parts in columns year, month and day. Any other
    columns are disregarded.

Returns: Logical vector with the same length as argument ymd has rows with TRUE for valid
days and FALSE for invalid days, or NA where the row in argument ymd has NA values.

Method is_gregorian_date(): Indicate which of the supplied dates are in the Gregorian part
of the calendar, e.g. 1582-10-15 or after.

Usage:

```
CFCalendarStandard$is_gregorian_date(ymd)
```

Arguments:

ymd data.frame with dates parsed into their parts in columns year, month and day. Any other
    columns are disregarded.

Returns: Logical vector with the same length as argument ymd has rows with TRUE for days in
the Gregorian part of the calendar and FALSE otherwise, or NA where the row in argument ymd
has NA values.

Method POSIX_compatible(): Indicate if the time series described using this calendar can be
safely converted to a standard date-time type (POSIXct, POSIXlt, Date). This is only the case
if all offsets are for timestamps fall on or after the start of the Gregorian calendar, 1582-10-15
00:00:00.

Usage:

```
CFCalendarStandard$POSIX_compatible(offsets)
```

Arguments:

offsets The offsets from the CFtime instance.

Returns: TRUE.

Method month_days(): Determine the number of days in the month of the calendar.

Usage:

```
CFCalendarStandard$month_days(ymd = NULL)
```

Arguments:

ymd data.frame, optional, with dates parsed into their parts.

Returns: A vector indicating the number of days in each month for the dates supplied as
argument ymd. If no dates are supplied, the number of days per month for the calendar as a
vector of length 12, for a regular year without a leap day.

Method leap_year(): Indicate which years are leap years.

Usage:

```
CFCalendarStandard$leap_year(yr)
```

Arguments:

yr Integer vector of years to test.

Returns: Logical vector with the same length as argument yr. NA is returned where elements in argument yr are NA.

Method `date2offset()`: Calculate difference in days between a `data.frame` of time parts and the origin.

Usage:

`CFCalendarStandard$date2offset(x)`

Arguments:

x `data.frame`. Dates to calculate the difference for.

Returns: Integer vector of a length equal to the number of rows in argument x indicating the number of days between x and the origin of the calendar, or NA for rows in x with NA values.

Method `offset2date()`: Calculate date parts from day differences from the origin. This only deals with days as these are impacted by the calendar. Hour-minute-second timestamp parts are handled in [CFCalendar](#).

Usage:

`CFCalendarStandard$offset2date(x)`

Arguments:

x Integer vector of days to add to the origin.

Returns: A `data.frame` with columns 'year', 'month' and 'day' and as many rows as the length of vector x.

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

`CFCalendarStandard$clone(deep = FALSE)`

Arguments:

deep Whether to make a deep clone.

CFCalendarTAI *International Atomic Time CF calendar*

Description

This class represents a calendar based on the International Atomic Time. Validity is from 1958 onwards, with dates represented using the Gregorian calendar. Given that this "calendar" is based on a universal clock, the concepts of leap second, time zone and daylight savings time do not apply.

Super classes

[CFtime::CFCalendar](#) -> [CFtime::CFCalendarProleptic](#) -> CFCalendarTAI

Methods

Public methods:

- [CFCalendarTAI$valid_days()](#)
- [CFCalendarTAI$clone()](#)

Method `valid_days()`: Indicate which of the supplied dates are valid.

Usage:

CFCalendarTAI$valid_days(ymd)

Arguments:

ymd data.frame with dates parsed into their parts in columns year, month and day. If present, the tz column is checked for illegal time zone offsets. Any other columns are disregarded.

Returns: Logical vector with the same length as argument ymd has rows with TRUE for valid days and FALSE for invalid days, or NA where the row in argument ymd has NA values.

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

CFCalendarTAI$clone(deep = FALSE)

Arguments:

deep Whether to make a deep clone.

CFCalendarUTC   *Coordinated Universal Time CF calendar*

Description

This class represents a calendar based on the Coordinated Universal Time. Validity is from 1972 onwards, with dates represented using the Gregorian calendar, up to the present (so future timestamps are not allowed). Leap seconds are considered in all calculations. Also, time zone information is irrelevant and may not be given.

In general, the calendar should use a unit of time of a second. Minute, hour and day are allowed but discouraged. Month and year as time unit are not allowed as there is no practical way to maintain leap second accuracy.

Super classes

[CFtime::CFCalendar](#) -> [CFtime::CFCalendarProleptic](#) -> CFCalendarUTC

Methods

Public methods:

- CFCalendarUTC$new()
- CFCalendarUTC$valid_days()
- CFCalendarUTC$parse()
- CFCalendarUTC$offsets2time()
- CFCalendarUTC$clone()

Method new(): Create a new CF UTC calendar.

Usage:

CFCalendarUTC$new(nm, definition)

Arguments:

nm The name of the calendar. This must be "utc".

definition The string that defines the units and the origin, as per the CF Metadata Conventions.

Method valid_days(): Indicate which of the supplied dates are valid.

Usage:

CFCalendarUTC$valid_days(ymd)

Arguments:

ymd data.frame with dates parsed into their parts in columns year, month and day. Any other columns are disregarded.

Returns: Logical vector with the same length as argument ymd has rows with TRUE for valid days and FALSE for invalid days, or NA where the row in argument ymd has NA values.

Method parse(): Parsing a vector of date-time character strings into parts. This includes any leap seconds. Time zone indications are not allowed.

Usage:

CFCalendarUTC$parse(d)

Arguments:

d character. A character vector of date-times.

Returns: A data.frame with columns year, month, day, hour, minute, second, time zone, and offset. Invalid input data will appear as NA. Note that the time zone is always "+0000" and is included to maintain compatibility with results from other calendars.

Method offsets2time(): Decompose a vector of offsets, in units of the calendar, to their timestamp values. This adds a specified amount of time to the origin of a CFTime object.

Usage:

CFCalendarUTC$offsets2time(offsets = NULL)

Arguments:

offsets Vector of numeric offsets to add to the origin of the calendar.

Returns: A data.frame with columns for the timestamp elements and as many rows as there are offsets.

Method clone(): The objects of this class are cloneable with this method.

Usage:

CFCalendarUTC$clone(deep = FALSE)

Arguments:

deep Whether to make a deep clone.

CFClimatology   *CFClimatology class*

Description

This class represents a climatological time coordinate, both its "calendar" and its "climatology" bounds values.

The "calendar" portion is managed by the CFTime class, from which this class is inherited. This class implements the specific behaviour of the climatological bounds values and includes methods to query the structure of the climatological data.

Super class

CFtime::CFTime -> CFClimatology

Active bindings

period (read-only) Character string indicating the period during the year over which the climatology was calculated.

years (read-only) Vector of two integer values indicating the years over which the climatology was calculated.

friendlyClassName Character string with class name for display purposes.

Methods

Public methods:

- CFClimatology$new()
- CFClimatology$print()
- CFClimatology$clone()

Method new(): Create a new instance of this class.

Usage:

CFClimatology$new(definition, calendar = "standard", offsets, bounds)

Arguments:

definition Character string of the units and origin of the calendar.

calendar Character string of the calendar to use. Must be one of the values permitted by the CF Metadata Conventions. If NULL, the "standard" calendar will be used.

offsets Numeric or character vector. When numeric, a vector of offsets from the origin in the time series. When a character vector of length 2 or more, timestamps in ISO8601 or UDUNITS format.

bounds The climatological bounds for the offsets. A `matrix` with columns for `offsets` and low values in the first row, high values in the second row. The bounds will oftentimes overlap or be discontinuous.

Method `print()`: Print a summary of the `CFClimatology` object to the console.

Usage:
`CFClimatology$print(...)`

Arguments:

`...` Ignored.

Returns: `self` invisibly.

Method `clone()`: The objects of this class are cloneable with this method.

Usage:
`CFClimatology$clone(deep = FALSE)`

Arguments:

deep Whether to make a deep clone.

References

https://cfconventions.org/Data/cf-conventions/cf-conventions-1.12/cf-conventions.html#climatological-statistics

---

CFfactor *Create a factor from the offsets in a* CFTime *instance*

Description

With this function a factor can be generated for the time series, or a part thereof, contained in the CFTime instance. This is specifically interesting for creating factors from the date part of the time series that aggregate the time series into longer time periods (such as month) that can then be used to process daily CF data sets using, for instance, `tapply()`.

Usage

```
CFfactor(t, period = "month", era = NULL)
```

Arguments

t	An instance of the `CFTime` class whose offsets will be used to construct the factor.
period	character. A character string with one of the values "year", "season", "quarter", "month" (the default), "dekad" or "day".
era	numeric or list, optional. Vector of years for which to construct the factor, or a list whose elements are each a vector of years. If era is not specified, the factor will use the entire time series for the factor.

Details

The factor will respect the calendar that the time series is built on. For periods longer than a day this will result in a factor where the calendar is no longer relevant (because calendars impacts days, not dekads, months, quarters, seasons or years).

The factor will be generated in the order of the offsets of the CFTime instance. While typical CF-compliant data sources use ordered time series there is, however, no guarantee that the factor is ordered as multiple CFTime objects may have been merged out of order. For most processing with a factor the ordering is of no concern.

If the era parameter is specified, either as a vector of years to include in the factor, or as a list of such vectors, the factor will only consider those values in the time series that fall within the list of years, inclusive of boundary values. Other values in the factor will be set to NA. The years need not be contiguous, within a single vector or among the list items, or in order.

The following periods are supported by this function:

* year, the year of each offset is returned as "YYYY".

* season, the meteorological season of each offset is returned as "Sx", with x being 1-4, pre-ceeded by "YYYY" if no era is specified. Note that December dates are labeled as belonging to the subsequent year, so the date "2020-12-01" yields "2021S1". This implies that for stan-dard CMIP files having one or more full years of data the first season will have data for the first two months (January and February), while the final season will have only a single month of data (December).

* quarter, the calendar quarter of each offset is returned as "Qx", with x being 1-4, preceeded by "YYYY" if no era is specified.

* month, the month of each offset is returned as "01" to "12", preceeded by "YYYY-" if no era is specified. This is the default period.

* dekad, ten-day periods are returned as "Dxx", where xx runs from "01" to "36", preceeded by "YYYY" if no era is specified. Each month is subdivided in dekads as follows: 1- days 01 - 10; 2- days 11 - 20; 3- remainder of the month.

* day, the month and day of each offset are returned as "MM-DD", preceeded by "YYYY-" if no era is specified.

It is not possible to create a factor for a period that is shorter than the temporal resolution of the source data set from which the t argument derives. As an example, if the source data set has monthly data, a dekad or day factor cannot be created.

Creating factors for other periods is not supported by this function. Factors based on the timestamp information and not dependent on the calendar can trivially be constructed from the output of the [as_timestamp()](#) function.

For non-era factors the attribute 'CFTime' of the result contains a CFTime instance that is valid for the result of applying the factor to a data set that the t argument is associated with. In other words, if CFTime instance 'At' describes the temporal dimension of data set 'A' and a factor 'Af' is generated like Af <- CFfactor(At), then Bt <- attr(Af, "CFTime") describes the temporal dimension of the result of, say, B <- apply(A, 1:2, tapply, Af, FUN). The 'CFTime' attribute is NULL for era factors.

Value

If era is a single vector or not specified, a factor with a length equal to the number of offsets in t. If era is a list, a list with the same number of elements and names as era, each containing a factor. Elements in the factor will be set to NA for time series values outside of the range of specified years.

The factor, or factors in the list, have attributes 'period', 'era' and 'CFTime'. Attribute 'period' holds the value of the period argument. Attribute 'era' indicates the number of years that are included in the era, or -1 if no era is provided. Attribute 'CFTime' holds an instance of CFTime that has the same definition as t, but with offsets corresponding to the mid-point of non-era factor levels; if the era argument is specified, attribute 'CFTime' is NULL.

See Also

[cut()](cut()) creates a non-era factor for arbitrary cut points.

Examples

```
t <- CFtime("days since 1949-12-01", "360_day", 19830:54029)

# Create a dekad factor for the whole time series
f <- CFfactor(t, "dekad")

# Create three monthly factors for early, mid and late 21st century eras
ep <- CFfactor(t, era = list(early = 2021:2040, mid = 2041:2060, late = 2061:2080))
```

CFfactor_coverage   *Coverage of time elements for each factor level*

Description

This function calculates the number of time elements, or the relative coverage, in each level of a factor generated by CFfactor().

Usage

```
CFfactor_coverage(t, f, coverage = "absolute")
```

Arguments

t	An instance of CFTime.
f	factor or list. A factor or a list of factors derived from the parameter t. The factor or list thereof should generally be generated by the function CFfactor().
coverage	"absolute" or "relative".

Value

If f is a factor, a numeric vector with a length equal to the number of levels in the factor, indicating the number of units from the time series in t contained in each level of the factor when coverage = "absolute" or the proportion of units present relative to the maximum number when coverage = "relative". If f is a list of factors, a list with each element a numeric vector as above.

Examples

```
t <- CFtime("days since 2001-01-01", "365_day", 0:364)
f <- CFfactor(t, "dekad")
CFfactor_coverage(t, f, "absolute")
```

CFfactor_units   *Number of base time units in each factor level*

Description

Given a factor as returned by CFfactor() and the CFTime instance from which the factor was derived, this function will return a numeric vector with the number of time units in each level of the factor.

Usage

```
CFfactor_units(t, f)
```

Arguments

t	An instance of CFTime.
f	A factor or a list of factors derived from the parameter t. The factor or list thereof should generally be generated by the function CFfactor().

Details

The result of this function is useful to convert between absolute and relative values. Climate change anomalies, for instance, are usually computed by differencing average values between a future period and a baseline period. Going from average values back to absolute values for an aggregate period (which is typical for temperature and precipitation, among other variables) is easily done with the result of this function, without having to consider the specifics of the calendar of the data set.

If the factor f is for an era (e.g. spanning multiple years and the levels do not indicate the specific year), then the result will indicate the number of time units of the period in a regular single year. In other words, for an era of 2041-2060 and a monthly factor on a standard calendar with a days unit, the result will be c(31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31). Leap days are thus only considered for the 366_day and all_leap calendars.

Note that this function gives the number of time units in each level of the factor - the actual number of data points in the cf instance per factor level may be different. Use CFfactor_coverage() to determine the actual number of data points or the coverage of data points relative to the factor level.

Value

If f is a factor, a numeric vector with a length equal to the number of levels in the factor, indicating the number of time units in each level of the factor. If f is a list of factors, a list with each element a numeric vector as above.

Examples

```
t <- CFtime("days since 2001-01-01", "365_day", 0:364)
f <- CFfactor(t, "dekad")
CFfactor_units(t, f)
```

CFTime   *CFTime class*

Description

This class manages the "time" dimension of netCDF files that follow the CF Metadata Conventions, and its productive use in R.

The class has a field cal which holds a specific calendar from the allowed types (all named calendars are supported). The calendar is also implemented as a (hidden) class which converts netCDF file encodings to timestamps as character strings, and vice-versa. Bounds information (the period of time over which a timestamp is valid) is used when defined in the netCDF file.

Additionally, this class has functions to ease use of the netCDF "time" information when processing data from netCDF files. Filtering and indexing of time values is supported, as is the generation of factors.

Public fields

cal The calendar of this CFTime instance, a descendant of the [CFCalendar](#) class.

offsets A numeric vector of offsets from the origin of the calendar.

resolution The average number of time units between offsets.

Active bindings

calendar (read-only) The calendar of this CFTime instance, a descendant of the [CFCalendar](#) class.

unit (read-only) The unit string of the calendar and time series.

length (read-only) Retrieve the number of offsets in the time series.

bounds Retrieve or set the bounds for the offsets. On setting, a matrix with columns for offsets and low values in the first row, high values in the second row. This may be simply TRUE to set regular and consecutive bounds.

friendlyClassName Character string with class name for display purposes.

Methods

Public methods:

- [CFTime$new()](#)
- [CFTime$print()](#)
- [CFTime$range()](#)
- [CFTime$as_timestamp()](#)
- [CFTime$format()](#)

- `CFTime$indexOf()`
- `CFTime$get_bounds()`
- `CFTime$set_bounds()`
- `CFTime$equidistant()`
- `CFTime$slice()`
- `CFTime$POSIX_compatible()`
- `CFTime$cut()`
- `CFTime$factor()`
- `CFTime$factor_units()`
- `CFTime$factor_coverage()`
- `CFTime$copy()`
- `CFTime$subset()`
- `CFTime$clone()`

Method `new()`: Create a new instance of this class.

Usage:
`CFTime$new(definition, calendar = "standard", offsets = NULL)`

Arguments:

definition Character string of the units and origin of the calendar.

calendar Character string of the calendar to use. Must be one of the values permitted by the CF Metadata Conventions. If `NULL`, the "standard" calendar will be used.

offsets Numeric or character vector, optional. When numeric, a vector of offsets from the origin in the time series. When a character vector of length 2 or more, timestamps in ISO8601 or UDUNITS format.

Method `print()`: Print a summary of the `CFTime` object to the console.

Usage:
`CFTime$print(...)`

Arguments:

`...` Ignored.

Returns: Self, invisibly.

Method `range()`: This method returns the first and last timestamp of the time series as a vector. Note that the offsets do not have to be sorted.

Usage:
`CFTime$range(format = "", bounds = FALSE)`

Arguments:

format Value of "date" or "timestamp". Optionally, a character string that specifies an alternate format.

bounds Logical to indicate if the extremes from the bounds should be used, if set. Defaults to `FALSE`.

Returns: Vector of two character strings that represent the starting and ending timestamps in the time series. If a `format` is supplied, that format will be used. Otherwise, if all of the timestamps in the time series have a time component of `00:00:00` the date of the timestamp is returned, otherwise the full timestamp (without any time zone information).

Method `as_timestamp()`: This method generates a vector of character strings or `POSIXcts` that represent the date and time in a selectable combination for each offset.

The character strings use the format `YYYY-MM-DDThh:mm:ss±hhmm`, depending on the `format` specifier. The date in the string is not necessarily compatible with `POSIXt` - in the `360_day` calendar `2017-02-30` is valid and `2017-03-31` is not.

For the "proleptic_gregorian" calendar the output can also be generated as a vector of `POSIXct` values by specifying `asPOSIX = TRUE`. The same is possible for the "standard" and "gregorian" calendars but only if all timestamps fall on or after 1582-10-15. If `asPOSIX = TRUE` is specified while the calendar does not support it, an error will be generated.

Usage:
`CFTime$as_timestamp(format = NULL, asPOSIX = FALSE)`

Arguments:

`format` character. A character string with either of the values "date" or "timestamp". If the argument is not specified, the format used is "timestamp" if there is time information, "date" otherwise.

`asPOSIX` logical. If `TRUE`, for "standard", "gregorian" and "proleptic_gregorian" calendars the output is a vector of `POSIXct` - for other calendars an error will be thrown. Default value is `FALSE`.

Returns: A character vector where each element represents a moment in time according to the `format` specifier.

Method `format()`: Format timestamps using a specific format string, using the specifiers defined for the [`base::strptime()`](base::strptime()) function, with limitations. The only supported specifiers are `bBdeFhHImMpRSTYz%`. Modifiers `E` and `O` are silently ignored. Other specifiers, including their percent sign, are copied to the output as if they were adorning text.

The formatting is largely oblivious to locale. The reason for this is that certain dates in certain calendars are not POSIX-compliant and the system functions necessary for locale information thus do not work consistently. The main exception to this is the (abbreviated) names of months (bB), which could be useful for pretty printing in the local language. For separators and other locale-specific adornments, use local knowledge instead of depending on system locale settings; e.g. specify %m/%d/%Y instead of %D.

Week information, including weekday names, is not supported at all as a "week" is not defined for non-standard CF calendars and not generally useful for climate projection data. If you are working with observed data and want to get pretty week formats, use the [`as_timestamp()`](as_timestamp()) method to generate `POSIXct` timestamps (observed data generally uses a "standard" calendar) and then use the [`base::format()`](base::format()) function which supports the full set of specifiers.

Usage:
`CFTime$format(format, usetz = FALSE)`

Arguments:

`format` A character string with `strptime` format specifiers. If omitted, the most economical format will be used: a full timestamp when time information is available, a date otherwise.

usetz Logical. Should the time zone offset be appended to the output? This is always in numerical form, i.e. "-0800", from UTC. Default is FALSE.

Returns: A vector of character strings with a properly formatted timestamp. Any format specifiers not recognized or supported will be returned verbatim.

Method `indexOf()`: Find the index in the time series for each timestamp given in argument x. Alternatively, when x is a numeric vector of index values, return the valid indices of the same vector, with the side effect being the attribute "CFTime" associated with the result.

Matching also returns index values for timestamps that fall between two elements of the time series - this can lead to surprising results when time series elements are positioned in the middle of an interval (as the CF Metadata Conventions instruct us to "reasonably assume"): a time series of days in January would be encoded in a netCDF file as `c("2024-01-01 12:00:00", "2024-01-02 12:00:00", "2024-01-03 12:00:00", ...)` so `x <- c("2024-01-01", "2024-01-02", "2024-01-03")` would result in `(NA, 1, 2)` (or `(NA, 1.5, 2.5)` with `method = "linear"`) because the date values in x are at midnight. This situation is easily avoided by ensuring that this `CFTime` instance has bounds set (use `bounds(y) <- TRUE` as a proximate solution if bounds are not stored in the netCDF file).

If bounds are set, the indices are informed by those bounds. If the bounds are not contiguous, returned values may be NA even if the value of x falls between two valid timestamps.

Values of x that are not valid timestamps according to the calendar of this `CFTime` instance will be returned as NA.

Argument x can also be a numeric vector of index values, in which case the valid values in x are returned. If negative values are passed, the positive counterparts will be excluded and then the remainder returned. Positive and negative values may not be mixed. Using a numeric vector has the side effect that the result has the attribute "CFTime" describing the temporal dimension of the slice. If index values outside of the range of `self` are provided, an error will be thrown.

Usage:
```
CFTime$indexOf(x, method = "constant", rightmost.closed = FALSE)
```

Arguments:

x Vector of `character`, `POSIXt` or `Date` values to find indices for, or a `numeric` vector.

method Single value of "constant" or "linear". If `"constant"`, return the index value for each match. If `"linear"`, return the index value with any fractional value.

rightmost.closed Whether or not to include the upper limit of argument x. Default is FALSE. This argument is ignored when argument x contains index values.

Returns: A numeric vector giving indices into `self` for the valid values of x. If there is at least 1 valid index, then attribute "CFTime" contains an instance of `CFTime` that describes the dimension of filtering the dataset associated with `self` with the result of this method, excluding any NA values.

Method `get_bounds()`: Return boundary values.

Usage:
```
CFTime$get_bounds(format)
```

Arguments:

format A string specifying a format for output, optional.

Returns: An array with dims(2, length(offsets)) with values for the boundaries. NULL if the boundaries have not been set.

Method set_bounds(): Set or delete the boundary values of the CFTime instance.

Usage:

CFTime$set_bounds(value)

Arguments:

value The boundary values to set, in units of the offsets. A matrix (2, length(self$offsets)). If NULL, the boundaries are deleted. If TRUE, make regular, consecutive boundaries.

Returns: Self, invisibly.

Method equidistant(): This method returns TRUE if the time series has uniformly distributed time steps between the extreme values, FALSE otherwise. First test without sorting; this should work for most data sets. If not, only then offsets are sorted. For most data sets that will work but for implied resolutions of month, season, year, etc based on a "days" or finer calendar unit this will fail due to the fact that those coarser units have a variable number of days per time step, in all calendars except for 360_day. For now, an approximate solution is used that should work in all but the most non-conformal exotic arrangements.

Usage:

CFTime$equidistant()

Returns: TRUE if all time steps are equidistant, FALSE otherwise, or NA if no offsets have been set.

Method slice(): Given a vector of character timestamps, return a logical vector of a length equal to the number of time steps in the time series with values TRUE for those time steps that fall between the two extreme values of the vector values, FALSE otherwise.

Usage:

CFTime$slice(extremes, closed = FALSE)

Arguments:

extremes Character vector of timestamps that represent the time period of interest. The extreme values are selected. Badly formatted timestamps are silently dropped.

closed Is the right side closed, i.e. included in the result? Default is FALSE. A specification of c("2022-01-01", "2023-01-01) will thus include all time steps that fall in the year 2022 when closed = FALSE but include 2023-01-01 if that exact value is present in the time series.

Returns: A logical vector with a length equal to the number of time steps in self with values TRUE for those time steps that fall between the extreme values, FALSE otherwise.
An attribute 'CFTime' will have the same definition as self but with offsets corresponding to the time steps falling between the two extremes. If there are no values between the extremes, the attribute is not set.

Method POSIX_compatible(): Can the time series be converted to POSIXt?

Usage:

CFTime$POSIX_compatible()

Returns: TRUE if the calendar support conversion to POSIXt, FALSE otherwise.

Method cut(): Create a factor for a CFTime instance.

When argument breaks is one of "year", "season", "quarter", "month", "dekad", "day", a factor is generated like by CFfactor(). When breaks is a vector of character timestamps a factor is produced with a level for every interval between timestamps. The last timestamp, therefore, is only used to close the interval started by the pen-ultimate timestamp - use a distant timestamp (e.g. range(x)[2]) to ensure that all offsets to the end of the CFTime time series are included, if so desired. The last timestamp will become the upper bound in the CFTime instance that is returned as an attribute to this function so a sensible value for the last timestamp is advisable.

This method works similar to base::cut.POSIXt() but there are some differences in the arguments: for breaks the set of options is different and no preceding integer is allowed, labels are always assigned using values of breaks, and the interval is always left-closed.

Usage:

CFTime$cut(breaks)

Arguments:

breaks A character string of a factor period (see CFfactor() for a description), or a character vector of timestamps that conform to the calendar of x, with a length of at least 2. Timestamps must be given in ISO8601 format, e.g. "2024-04-10 21:31:43".

Returns: A factor with levels according to the breaks argument, with attributes 'period', 'era' and 'CFTime'. When breaks is a factor period, attribute 'period' has that value, otherwise it is '"day"'. When breaks is a character vector of timestamps, attribute 'CFTime' holds an instance of CFTime that has the same definition as x, but with (ordered) offsets generated from the breaks. Attribute 'era' is always -1.

Method factor(): Generate a factor for the offsets, or a part thereof. This is specifically interesting for creating factors from the date part of the time series that aggregate the time series into longer time periods (such as month) that can then be used to process daily CF data sets using, for instance, tapply().

The factor will respect the calendar that the time series is built on.

The factor will be generated in the order of the offsets. While typical CF-compliant data sources use ordered time series there is, however, no guarantee that the factor is ordered. For most processing with a factor the ordering is of no concern.

If the era parameter is specified, either as a vector of years to include in the factor, or as a list of such vectors, the factor will only consider those values in the time series that fall within the list of years, inclusive of boundary values. Other values in the factor will be set to NA.

The following periods are supported by this method:

- year, the year of each offset is returned as "YYYY".
- season, the meteorological season of each offset is returned as "Sx", with x being 1-4, preceeded by "YYYY" if no era is specified. Note that December dates are labeled as belonging to the subsequent year, so the date "2020-12-01" yields "2021S1". This implies that for standard CMIP files having one or more full years of data the first season will have data for the first two months (January and February), while the final season will have only a single month of data (December).
- quarter, the calendar quarter of each offset is returned as "Qx", with x being 1-4, preceeded by "YYYY" if no era is specified.
- month, the month of each offset is returned as "01" to "12", preceeded by "YYYY-" if no era is specified. This is the default period.

- dekad, ten-day periods are returned as "Dxx", where xx runs from "01" to "36", preceeded by "YYYY" if no era is specified. Each month is subdivided in dekads as follows: 1- days 01 - 10; 2- days 11 - 20; 3- remainder of the month.
- day, the month and day of each offset are returned as "MM-DD", preceeded by "YYYY-" if no era is specified.

It is not possible to create a factor for a period that is shorter than the temporal resolution of the calendar. As an example, if the calendar has a monthly unit, a dekad or day factor cannot be created.

Creating factors for other periods is not supported by this method. Factors based on the timestamp information and not dependent on the calendar can trivially be constructed from the output of the `as_timestamp()` function.

Attribute 'CFTime' of the result contains a `CFTime` instance that is valid for the result of applying the factor to a resource that this instance is associated with. In other words, if `CFTime` instance 'At' describes the temporal dimension of resource 'A' and a factor 'Af' is generated from `Af <- At$factor()`, then `Bt <- attr(Af, "CFTime")` describes the temporal dimension of the result of, say, `B <- apply(A, 1:2, tapply, Af, FUN)`. The 'CFTime' attribute contains a CFClimatology instance for era factors.

Usage:
`CFTime$factor(period = "month", era = NULL)`

Arguments:

`period` character. A character string with one of the values "year", "season", "quarter", "month" (the default), "dekad" or "day".

`era` numeric or list, optional. Vector of years for which to construct the factor, or a list whose elements are each a vector of years. The extreme values of the supplied vector will be used. Note that if a single year is specified that the result is valid, but not a climatological statistic. If era is not specified, the factor will use the entire time series for the factor.

Returns: If era is a single vector or NULL, a factor with a length equal to the number of offsets in this instance. If era is a list, a list with the same number of elements and names as era, each containing a factor. Elements in the factor will be set to NA for time series values outside of the range of specified years.

The factor, or factors in the list, have attributes 'period', 'era' and 'CFTime'. Attribute 'period' holds the value of the `period` argument. Attribute 'era' indicates the number of years that are included in the era, or -1 if no era is provided. Attribute 'CFTime' holds an instance of `CFTime` or `CFClimatology` that has the same definition as this instance, but with offsets corresponding to the mid-point of the factor levels.

Method `factor_units()`: Given a factor as produced by `CFTime$factor()`, this method will return a numeric vector with the number of time units in each level of the factor.

The result of this method is useful to convert between absolute and relative values. Climate change anomalies, for instance, are usually computed by differencing average values between a future period and a baseline period. Going from average values back to absolute values for an aggregate period (which is typical for temperature and precipitation, among other variables) is easily done with the result of this method, without having to consider the specifics of the calendar of the data set.

If the factor `f` is for an era (e.g. spanning multiple years and the levels do not indicate the specific year), then the result will indicate the number of time units of the period in a regular single year.

In other words, for an era of 2041-2060 and a monthly factor on a standard calendar with a days unit, the result will be c(31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31). Leap days are thus only considered for the 366_day and all_leap calendars.

Note that this function gives the number of time units in each level of the factor - the actual number of data points in the time series per factor level may be different. Use CFfactor_coverage() to determine the actual number of data points or the coverage of data points relative to the factor level.

Usage:
CFTime$factor_units(f)

Arguments:
f A factor or a list of factors derived from the method CFTime$factor().

Returns: If f is a factor, a numeric vector with a length equal to the number of levels in the factor, indicating the number of time units in each level of the factor. If f is a list of factors, a list with each element a numeric vector as above.

Method factor_coverage(): Calculate the number of time elements, or the relative coverage, in each level of a factor generated by CFTime$factor().

Usage:
CFTime$factor_coverage(f, coverage = "absolute")

Arguments:
f A factor or a list of factors derived from the method CFTime$factor().
coverage "absolute" or "relative".

Returns: If f is a factor, a numeric vector with a length equal to the number of levels in the factor, indicating the number of units from the time series contained in each level of the factor when coverage = "absolute" or the proportion of units present relative to the maximum number when coverage = "relative". If f is a list of factors, a list with each element a numeric vector as above.

Method copy(): Create a copy of the current instance. The copy is completely separate from the current instance.

Usage:
CFTime$copy()

Returns: A new CFTime instance with identical definition and set of timestamps.

Method subset(): Get a new CFTime instance that is a subset of the current instance, including any boundary values.

Usage:
CFTime$subset(rng)

Arguments:
rng The numeric range of indices to subset this instance to.

Returns: A new CFTime instance with identical definition and set of timestamps according to the rng argument.

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
CFTime$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

References

https://cfconventions.org/Data/cf-conventions/cf-conventions-1.12/cf-conventions.html#time-coordinate

---

CFtime-function   *Create a CFTime object*

Description

This function creates an instance of the [CFTime](#) class. The arguments to the call are typically read from a CF-compliant data file with climatological observations or climate projections. Specification of arguments can also be made manually in a variety of combinations.

Usage

```
CFtime(definition, calendar = "standard", offsets, from, to, by, length.out)
```

Arguments

definition	A character string describing the time coordinate.
calendar	A character string describing the calendar to use with the time dimension definition string. Default value is "standard".
offsets	Numeric or character vector, optional. When numeric, a vector of offsets from the origin in the time series. When a character vector, timestamps in ISO8601 or UDUNITS format.
from, to	Optional. Character timestamps in ISO8601 or UDUNITS format. When `from` is given, a sequence of timestamps is generated with `from` as the starting timestamp. Either argument `to` or `length.out` must be provided as well. Ignored when argument `offsets` is not NULL.
by	Optional. A single character string representing a time interval, composed of a number and a time unit separated by a space, such as in "6 hours". When argument `from` is supplied, argument by will be the separation between successive timestamps. Note that the time unit in the string does not have to be the same as the unit in the `definition` argument but it must be an allowable unit of time. Time interval units of "months" and "years" are strongly discouraged unless the same time unit is used in the `definition` argument - in all other cases there is a loss of precision due to the ambiguity in the time units.
length.out	Optional. An numeric value that indicates the lengths of the time series to generate, rounded up if fractional. Ignored when argument `to` is provided.

Details

A time series can also be constructed like a sequence. In this case argument offsets should be
NULL or missing and arguments from and by provided, with either of arguments to or length.out
indicating the end of the time series. Arguments should be named to avoid ambiguity.

Value

An instance of the CFTime class.

Examples

```
# Using numeric offset values - this is how a netCDF file works
CFtime("days since 1850-01-01", "julian", 0:364)

# A time object with a single defined timestamp
CFtime("hours since 2023-01-01", "360_day", "2023-01-30T23:00")

# A time series from a sequence with an end point
CFtime("days since 2023-01-01", from = "2020-01-01", to = "2023-12-31", by = "12 days")

# A time series from a sequence with a specified length
CFtime("days since 2023-01-01", from = "2020-01-01T03:00:00", by = "6 hr", length.out = 31 * 4)
```

---

cut.CFTime   *Create a factor for a* CFTime *instance*

Description

Method for base::cut() applied to CFTime objects.

Usage

```
## S3 method for class 'CFTime'
cut(x, breaks, ...)
```

Arguments

x   An instance of CFTime.

breaks   A character string of a factor period (see CFfactor() for a description), or a
             character vector of timestamps that conform to the calendar of x, with a length
             of at least 2. Timestamps must be given in ISO8601 format, e.g. "2024-04-10
             21:31:43".

...   Ignored.

Details

When breaks is one of "year", "season", "quarter", "month", "dekad", "day" a factor is
generated like by CFfactor().

When breaks is a vector of character timestamps a factor is produced with a level for every interval
between timestamps. The last timestamp, therefore, is only used to close the interval started by the
pen-ultimate timestamp - use a distant timestamp (e.g. range(x)[2]) to ensure that all offsets to
the end of the CFTime time series are included, if so desired. The last timestamp will become the
upper bound in the CFTime instance that is returned as an attribute to this function so a sensible
value for the last timestamp is advisable.

This method works similar to base::cut.POSIXt() but there are some differences in the argu-
ments: for breaks the set of options is different and no preceding integer is allowed, labels are
always assigned using values of breaks, and the interval is always left-closed.

Value

A factor with levels according to the breaks argument, with attributes 'period', 'era' and 'CFTime'.
When breaks is a factor period, attribute 'period' has that value, otherwise it is '"day"'. When
breaks is a character vector of timestamps, attribute 'CFTime' holds an instance of CFTime that
has the same definition as x, but with (ordered) offsets generated from the breaks. Attribute 'era'
is always -1.

See Also

CFfactor() produces a factor for several fixed periods, including for eras.

Examples

```
x <- CFtime("days since 2021-01-01", "365_day", 0:729)
breaks <- c("2022-02-01", "2021-12-01", "2023-01-01")
cut(x, breaks)
```

---

definition   *Properties of a CFTime object*

Description

These functions return the properties of an instance of the CFTime class. The properties are all
read-only, but offsets can be added using the + operator.

Usage

```
definition(t)

calendar(t)

unit(t)
```

```
origin(t)

timezone(t)

offsets(t)

resolution(t)
```

Arguments

t   An instance of `CFTime`.

Value

`calendar()` and `unit()` return a character string. `origin()` returns a data frame of timestamp elements with a single row of data. `timezone()` returns the calendar time zone as a character string. `offsets()` returns a vector of offsets or `NULL` if no offsets have been set.

Functions

- `definition()`: The definition string of the `CFTime` instance.

- `calendar()`: The calendar of the `CFTime` instance.

- `unit()`: The unit of the `CFTime` instance.

- `origin()`: The origin of the `CFTime` instance in timestamp elements.

- `timezone()`: The time zone of the calendar of the `CFTime` instance as a character string.

- `offsets()`: The offsets of the `CFTime` instance as a numeric vector.

- `resolution()`: The average separation between the offsets in the `CFTime` instance.

Examples

```
t <- CFtime("days since 1850-01-01", "julian", 0:364)
definition(t)
calendar(t)
unit(t)
timezone(t)
origin(t)
offsets(t)
resolution(t)
```

---

format.CFTime   *Format the timestamps in the* CFTime *instance*

Description

The formatting is largely oblivious to locale. The reason for this is that certain dates in certain calendars are not POSIX-compliant and the system functions necessary for locale information thus do not work consistently. The main exception to this is the (abbreviated) names of months (bB), which could be useful for pretty printing in the local language. For separators and other locale-specific adornments, use local knowledge instead of depending on system locale settings; e.g. specify %m/%d/%Y instead of %D.

Usage

```
## S3 method for class 'CFTime'
format(x, format = "", tz = "", usetz = FALSE, ...)
```

Arguments

x	The CFTime instance whose timestamps will be formatted.
format	A character string. The default for the format methods is "%Y-%m-%dT%H:%M:%S" if any timestamp has a time component which is not midnight, and "%Y-%m-%d" otherwise. The only supported specifiers are bBdeFhHImMpRSTYz%. Modifiers E and O are silently ignored. Other specifiers, including their percent sign, are copied to the output as if they were adorning text.
tz	Ignored.
usetz	Logical. Should the time zone offset be appended to the output? This is always in numerical form, i.e. "-0800", from UTC.
...	Ignored.

Details

Week information, including weekday names, is not supported at all as a "week" is not defined for non-standard CF calendars and not generally useful for climate projection data. If you are working with observed data and want to get pretty week formats, use the as_timestamp() method to generate POSIXct timestamps (observed data generally uses a "standard" calendar) and then use the base::format() function which supports the full set of specifiers.

---

indexOf   *Find the index of timestamps in the time series*

Description

Find the index in the time series for each timestamp given in argument x. Values of x that are before the earliest value or after the latest value in y will be returned as NA. Alternatively, when x is a numeric vector of index values, return the valid indices of the same vector, with the side effect being the attribute "CFTime" associated with the result.

Usage

```
indexOf(x, y, method = "constant", rightmost.closed = FALSE)
```

Arguments

x	Vector of `character`, `POSIXt` or `Date` values to find indices for, or a numeric vector.
y	A CFTime instance.
method	Single value of "constant" or "linear". If `"constant"` or when bounds are set on argument y, return the index value for each match. If `"linear"`, return the index value with any fractional value.
rightmost.closed	
	Whether or not to include the upper limit of argument x. Default is `FALSE`. This argument is ignored when argument x contains index values.

Details

Timestamps can be provided as vectors of character strings, `POSIXct` or `Date`.

Matching also returns index values for timestamps that fall between two elements of the time series - this can lead to surprising results when time series elements are positioned in the middle of an interval (as the CF Metadata Conventions instruct us to "reasonably assume"): a time series of days in January would be encoded in a netCDF file as c("2024-01-01 12:00:00", "2024-01-02 12:00:00", "2024-01-03 12:00:00", ...) so x <- c("2024-01-01", "2024-01-02", "2024-01-03") would result in (NA, 1, 2) (or (NA, 1.5, 2.5) with method = "linear") because the date values in x are at midnight. This situation is easily avoided by ensuring that y has bounds set (use bounds(y) <- TRUE as a proximate solution if bounds are not stored in the netCDF file). See the Examples.

If bounds are set, the indices are informed by those bounds. If the bounds are not contiguous, returned values may be NA even if the value of x falls between two valid timestamps.

Values of x that are not valid timestamps according to the calendar of y will be returned as NA.

x can also be a numeric vector of index values, in which case the valid values in x are returned. If negative values are passed, the positive counterparts will be excluded and then the remainder returned. Positive and negative values may not be mixed. Using a numeric vector has the side effect that the result has the attribute "CFTime" describing the temporal dimension of the slice. If index values outside of the range of y (1:length(y)) are provided, an error will be thrown.

Value

A numeric vector giving indices into y for the values of x. If there is at least 1 valid index, then attribute "CFTime" contains an instance of CFTime that describes the dimension of filtering the dataset associated with y with the result of this method, excluding any NA values.

Examples

```
cf <- CFtime("days since 2020-01-01", "360_day", 1440:1799 + 0.5)
as_timestamp(cf)[1:3]
x <- c("2024-01-01", "2024-01-02", "2024-01-03")
indexOf(x, cf)
indexOf(x, cf, method = "linear")

bounds(cf) <- TRUE
```

```
indexOf(x, cf)

# Non-existent calendar day in a `360_day` calendar
x <- c("2024-03-30", "2024-03-31", "2024-04-01")
indexOf(x, cf)

# Numeric x
indexOf(c(29, 30, 31), cf)
```

---

is_complete   *Indicates if the time series is complete*

Description

This function indicates if the time series is complete, meaning that the time steps are equally spaced and there are thus no gaps in the time series.

Usage

```
is_complete(x)
```

Arguments

x   An instance of the [CFTime](#) class.

Details

This function gives exact results for time series where the nominal *unit of separation* between observations in the time series is exact in terms of the calendar unit. As an example, for a calendar unit of "days" where the observations are spaced a fixed number of days apart the result is exact, but if the same calendar unit is used for data that is on a monthly basis, the *assessment* is approximate because the number of days per month is variable and dependent on the calendar (the exception being the 360_day calendar, where the assessment is exact). The *result* is still correct in most cases (including all CF-compliant data sets that the developers have seen) although there may be esoteric constructions of CFTime and offsets that trip up this implementation.

Value

logical. TRUE if the time series is complete, with no gaps; FALSE otherwise. If no offsets have been added to the CFTime instance, NA is returned.

Examples

```
t <- CFtime("days since 1850-01-01", "julian", 0:364)
is_complete(t)
```

---

length.CFTime   *The length of the offsets contained in the* CFTime *instance.*

Description

The length of the offsets contained in the CFTime instance.

Usage

```
## S3 method for class 'CFTime'
length(x)
```

Arguments

x   The CFTime instance whose length will be returned

Value

The number of offsets in the specified CFTime instance.

Examples

```
t <- CFtime("days since 1850-01-01", "julian", 0:364)
length(t)
```

---

month_days   *Return the number of days in a month given a certain CF calendar*

Description

Given a vector of dates as strings in ISO 8601 or UDUNITS format and a CFTime object, this
function will return a vector of the same length as the dates, indicating the number of days in
the month according to the calendar specification. If no vector of days is supplied, the function
will return an integer vector of length 12 with the number of days for each month of the calendar
(disregarding the leap day for standard and julian calendars).

Usage

```
month_days(t, x = NULL)
```

Arguments

t   The CFTime instance to use.

x   character. An optional vector of dates as strings with format YYYY-MM-DD. Any
                     time part will be silently ingested.

Value

A vector indicating the number of days in each month for the vector of dates supplied as argument x. Invalidly specified dates will result in an NA value. If no dates are supplied, the number of days per month for the calendar as a vector of length 12.

See Also

When working with factors generated by CFfactor(), it is usually better to use CFfactor_units() as that will consider leap days for non-era factors. CFfactor_units() can also work with other time periods and calendar units, such as "hours per month", or "days per season".

Examples

```
dates <- c("2021-11-27", "2021-12-10", "2022-01-14", "2022-02-18")
t <- CFtime("days since 1850-01-01", "standard")
month_days(t, dates)

t <- CFtime("days since 1850-01-01", "360_day")
month_days(t, dates)

t <- CFtime("days since 1850-01-01", "all_leap")
month_days(t, dates)

month_days(t)
```

---

parse_timestamps   *Parse series of timestamps in CF format to date-time elements*

Description

This function will parse a vector of timestamps in ISO8601 or UDUNITS format into a data frame with columns for the elements of the timestamp: year, month, day, hour, minute, second, time zone. Those timestamps that could not be parsed or which represent an invalid date in the indicated CFtime instance will have NA values for the elements of the offending timestamp (which will generate a warning).

Usage

```
parse_timestamps(t, x)
```

Arguments

t	An instance of CFTime to use when parsing the date.
x	Vector of character strings representing timestamps in ISO8601 extended or UDUNITS broken format.

Details

The supported formats are the *broken timestamp* format from the UDUNITS library and ISO8601 *extended*, both with minor changes, as suggested by the CF Metadata Conventions. In general, the format is YYYY-MM-DD hh:mm:ss.sss hh:mm. The year can be from 1 to 4 digits and is interpreted literally, so 79-10-24 is the day Mount Vesuvius erupted and destroyed Pompeii, not 1979-10-24. The year and month are mandatory, all other fields are optional. There are defaults for all missing values, following the UDUNITS and CF Metadata Conventions. Leading zeros can be omitted in the UDUNITS format, but not in the ISO8601 format. The optional fractional part can have as many digits as the precision calls for and will be applied to the smallest specified time unit. In the result of this function, if the fraction is associated with the minute or the hour, it is converted into a regular hh:mm:ss.sss format, i.e. any fraction in the result is always associated with the second, rounded down to milli-second accuracy. The separator between the date and the time can be a single whitespace character or a T.

The time zone is optional and should have at least the hour or Z if present, the minute is optional. The time zone hour can have an optional sign. In the UDUNITS format the separator between the time and the time zone must be a single whitespace character, in ISO8601 there is no separation between the time and the timezone. Time zone names are not supported (as neither UDUNITS nor ISO8601 support them) and will cause parsing to fail when supplied, with one exception: the designator "UTC" is silently dropped (i.e. interpreted as "00:00").

Currently only the extended formats (with separators between the elements) are supported. The vector of timestamps may have any combination of ISO8601 and UDUNITS formats.

Value

A data.frame with constituent elements of the parsed timestamps in numeric format. The columns are year, month, day, hour, minute, second (with an optional fraction), time zone (character string), and the corresponding offset value from the origin. Invalid input data will appear as NA - if this is the case, a warning message will be displayed - other missing information on input will use default values.

Examples

```
t <- CFtime("days since 0001-01-01", "proleptic_gregorian")

# This will have `NA`s on output and generate a warning
timestamps <- c("2012-01-01T12:21:34Z", "12-1-23", "today",
                "2022-08-16T11:07:34.45-10", "2022-08-16 10.5+04")
parse_timestamps(t, timestamps)
```

---

range.CFTime   *Extreme time series values*

Description

Character representation of the extreme values in the time series.

Usage

```
## S3 method for class 'CFTime'
range(x, format = "", bounds = FALSE, ..., na.rm = FALSE)
```

Arguments

x	An instance of the [CFTime](#) class.
format	A character string with format specifiers, optional. If it is missing or an empty string, the most economical ISO8601 format is chosen: "date" when no time information is present in x, "timestamp" otherwise. Otherwise a suitable format specifier can be provided.
bounds	Logical to indicate if the extremes from the bounds should be used, if set. Defaults to FALSE.
...	Ignored.
na.rm	Ignored.

Value

Vector of two character representations of the extremes of the time series.

Examples

```
cf <- CFtime("days since 1850-01-01", "julian", 0:364)
range(cf)
range(cf, "%Y-%b-%e")
```

---

slice   *Which time steps fall within extreme values*

Description

Given a vector of character timestamps, return a logical vector of a length equal to the number of time steps in the time series with values TRUE for those time steps that fall between the two extreme values of the vector values, FALSE otherwise.

Usage

```
slice(x, extremes, rightmost.closed = FALSE)
```

Arguments

x	The CFTime instance to operate on.
extremes	Character vector of timestamps that represent the time period of interest. The extreme values are selected. Badly formatted timestamps are silently dropped.

rightmost.closed

> Is the right side closed, i.e. included in the result? Default is FALSE. A specifi-
> cation of c("2022-01-01", "2023-01-01) will thus include all time steps that
> fall in the year 2022 when closed = FALSE but include 2023-01-01 if that exact
> value is present in the time series.

Details

If bounds were set these will be preserved.

Value

A logical vector with a length equal to the number of time steps in x with values TRUE for those time
steps that fall between the extreme values, FALSE otherwise.

An attribute 'CFTime' will have the same definition as x but with offsets corresponding to the time
steps falling between the two extremes. If there are no values between the extremes, the attribute is
NULL.

Examples

```
t <- CFtime("hours since 2023-01-01 00:00:00", "standard", 0:23)
slice(t, c("2022-12-01", "2023-01-01 03:00"))
```

---

str.CFTime   *Compact display of a CFTime instance*

Description

Compact display of a CFTime instance

Usage

```
## S3 method for class 'CFTime'
str(object, ...)
```

Arguments

object   A CFTime instance.

...   Ignored.

[.CFTime   *Subset a* CFTime *instance by position in the time series*

Description

Subset a CFTime instance by position in the time series

Usage

```
## S3 method for class 'CFTime'
x[i = TRUE, ...]
```

Arguments

x	A CFTime instance.
i	A vector a positive integer values to indicate which values to extract from the time series by position. If negative values are passed, their positive counterparts will be excluded and then the remainder returned. Positive and negative values may not be mixed.
...	Ignored.

Value

A numeric vector with those values of i (or the inverse, when negative) that are valid in x. If there is at least 1 valid result, then attribute "CFTime" of the returned value contains an instance of CFTime that describes the dimension of filtering the dataset associated with x with the result of this function, excluding any NA values.

References

https://github.com/R-CF/CFtime/issues/20

Index