Package 'fame'

December 15, 2006

Title Interface for FAME time series database

Version 1.0

Author Jeff Hallman <jhallman@frb.gov>

Depends R (>= 2.0)

Description Includes FAME storage and retrieval function, as well as functions and S3 classes for time indexes and time indexed series, which are compatible with FAME frequencies.

Maintainer Jeff Hallman <jhallman@frb.gov>

License Public domain

R topics documented:

`addLast` *Add a function to be executed when R exits.*

Description

Add a function to be executed when R exits.

Usage

```
addLast(fun)
```

Arguments

`fun`	Function to be called.

Details

`addLast` defines `.Last` (if not already present) or redifines it so that the function `fun` will be called when R exits. The latter is accomplished by saving the current definition of `.Last` and creating a new `.Last` function that calls `fun` and then the original `.Last` function.

Value

None.

Author(s)

Gregory R. Warnes ⟨gregory.r.warnes@pfizer.com⟩

See Also

`.Last`

Examples

```
## Not run:
## Print a couple of cute messages when R exits.
helloWorld <- function() cat("\nHello World!\n")
byeWorld <- function() cat("\nGoodbye World!\n")

addLast(byeWorld)
addLast(helloWorld)

q("no")
```

```
## Should yield:
##
##   Save workspace image? [y/n/c]: n
##
##   Hello World!
##
##   Goodbye World!
##
##   Process R finished at Tue Nov 22 10:28:55 2005

## Unix-flavour example: send Rplots.ps to printer on exit.
myLast <- function()
{
  cat("Now sending PostScript graphics to the printer:\n")
  system("lpr Rplots.ps")
  cat("bye bye...\n")
}
addLast(myLast)
quit("yes")

## Should yield:
##
##  Now sending PostScript graphics to the printer:
##  lpr: job 1341 queued
##  bye bye...
##
##   Process R finished at Tue Nov 22 10:28:55 2005
## End(Not run)
```

`aggregate.tis` *Compute Summary Statistics of Time Series Subsets*

Description

Splits the data into subsets, computes summary statistics for each, and returns the result in a convenient form.

Usage

```
## S3 method for class 'tis':
aggregate(x, FUN = sum, ...)
## S3 method for class 'ts':
aggregate(x, nfrequency = 1, FUN = sum, ndeltat = 1,
          ts.eps = getOption("ts.eps"), ...)
```

Arguments

`x`	a `ts` or `tis` time series.

FUN	a scalar function to compute the summary statistics which can be applied to all data subsets.
nfrequency	new number of observations per unit of time; must be a divisor of the frequency of x.
ndeltat	new fraction of the sampling period between successive observations; must be a divisor of the sampling interval of x.
ts.eps	tolerance used to decide if nfrequency is a sub-multiple of the original frequency.
...	further arguments passed to or used by methods.

Details

These are time series methods for the generic aggregate function.

aggregate.ts has been reimplemented in package:frb to insure that the resulting time series starts on a boundary of the new frequency. Suppose, for example, that x is a monthly series starting in February 1990, nfrequency is 4, and FUN is mean. The package:frb implementation will return a series whose first observation is the average of the April, May and June observations of the input series, and the first element of it's tsp will be 1990.25. The quarters end in March, June, September and December. The first two monthly observations (February and March) are ignored because they don't span a quarter.

The package:stats implementation would return a quarterly series whose first observation is the average of the February, March and April monthly observations, and it's tsp will start with 1990.083, which corresponds to quarters ending in January, April, July and October. In our experience at the Fed, this is not the expected behavior.

aggregate.ts and aggregate.tis operate similarly. If x is not a time series, it is coerced to one. Then, the variables in x are split into appropriate blocks of length frequency(x) / nfrequency, and FUN is applied to each such block, with further (named) arguments in ... passed to it. The result returned is a time series with frequency nfrequency holding the aggregated values.

Author(s)

Jeff Hallman

See Also

apply, lapply, tapply, aggregate, and convert.

Examples

```
z <- tis(1:24, start = latestJanuary()) ## a monthly series
aggregate(z, nf = 4, FUN = mean)        ## quarterly average
aggregate(z, nf = 1, FUN = function(x) x[length(x)]) ## December is annual level
```

as.data.frame.tis *Coerce to a Data Frame*

Description

Coerce a Time Indexed Series to a data frame.

Usage

```
## S3 method for class 'tis':
as.data.frame(x, ...)
```

Arguments

x	a tis series
...	other args passed on to as.data.frame.matrix or as.data.frame.vector

Details

The function is very simple: it calls as.data.frame.matrix if x is a matrix, or as.data.frame.vector if it is not.

Value

a data frame.

See Also

data.frame

as.Date.jul *Convert ti or jul objects to Dates*

Description

Methods to convert ti and jul objects to class "Date" representing calendar dates.

Usage

```
## S3 method for class 'ti':
as.Date(x, ...)
## S3 method for class 'jul':
as.Date(x, ...)
```

Arguments

x	A ti or jul object to be converted.
...	Ignored.

Value

An object of class "Date".

See Also

as.Date for the generic function, Date for details of the date class.

Examples

```
as.Date(today())           ## invokes as.Date.ti
as.Date(jul(today() - 7)) ## a week ago, uses as.Date.jul
```

---

askForString *Ask User for a Password or String*

Description

Prompt the user for a password or a string. These functions interact with the user differently depending on the environment in which R is running.

Usage

```
askForString(prompt = "?: ", default = "")
askForPassword(prompt = "Password: ")
```

Arguments

prompt	Prompt to show when asking for user input
default	String shown as initial answer in dialog box or Emacs minibuffer, if either is available

Value

The password or string entered.

Note

If R is running under Emacs, the password prompt is shown in the minibuffer rather than in a dialog box in order to take advantage of the read-passwd function in Emacs, which does shows astericks as the password is entered.

Author(s)

Jeff Hallman

See Also

readline

---

as.matrix.tis *Create a Matrix from a Time Indexed Series*

Description

The function adds a dim attribute of c(length(x), 1) to its argument unless it already has a dim attribute of length 2.

Usage

```
## S3 method for class 'tis':
as.matrix(x)
```

Arguments

x	a tis object

Value

A tis object with a dim attribute of length 2.

Author(s)

Jeff Hallman

---

as.POSIXct.jul *Date-time Conversion Functions*

Description

Functions to create objects of classes "POSIXlt" and "POSIXct" representing calendar dates and times.

Usage

```
## S3 method for class 'jul':
as.POSIXct(x, ...)
## S3 method for class 'ti':
as.POSIXct(x, ...)
as.POSIXlt(x, tz = "", ...)
## Default S3 method:
as.POSIXlt(x, tz = "", ...)
## S3 method for class 'Date':
as.POSIXlt(x, ...)
## S3 method for class 'jul':
as.POSIXlt(x, ...)
## S3 method for class 'ti':
as.POSIXlt(x, ...)
## S3 method for class 'POSIXlt':
as.POSIXlt(x, tz = "", ...)
```

Arguments

`x`	An object to be converted.
`tz`	A timezone specification to be used for the conversion, *if one is required*. System-specific, but `""` is the current timezone, and `"GMT"` is UTC (Coordinated Universal Time, in French).
`...`	other args passed to underlying functions

Details

The `as.POSIX*` functions convert an object to one of the two classes used to represent date/times (calendar dates plus time to the nearest second). They can take convert a wide variety of objects, including objects of the other class and of classes `"Date"`, `"date"` (from package **date** or **survival**), `"chron"` and `"dates"` (from package **chron**) to these classes. Dates are treated as being at midnight UTC.

They can also convert character strings of the formats `"2001-02-03"` and `"2001/02/03"` optionally followed by white space and a time in the format `"14:52"` or `"14:52:03"`. (Formats such as `"01/02/03"` are ambiguous but can be converted via a format specification by `strptime`.)

Logical `NA`s can be converted to either of the classes, but no other logical vectors can be.

`asPOSIXlt` is generic, while the other functions documented here are methods for the generic `as.POSIX*` functions that convert objects to the `POSIXct` and `POSIXlt` classes.

Value

`as.POSIXct` and `as.POSIXlt` return objects of the appropriate class. If `tz` was specified, `as.POSIXlt` will give an appropriate `"tzone"` attribute.

Note

The standard `as.POSIXlt` function in `package:base` is not generic. That function has been renamed `as.POSIZlt.default` and replaced with a generic function calling `UseMethod`.

Author(s)

Jeff Hallman

See Also

`as.POSIXct` and `link{as.POSIXlt}` for the generic functions, and DateTimeClasses for details of the classes.

`as.ts.tis` *Convert a Time Indexed Series to a Time Series*

Description

Constructs a `ts` object from a `tis` object. The `tis` object's starting `year`, starting `cycle`, and `frequency`, along with the object's data, in a call to the `ts` function.

Usage

```
## S3 method for class 'tis':
as.ts(x, ...)
```

Arguments

`x`	a `tis` object to be converted
`...`	Ignored

Details

The `tis` class covers more frequencies than the `ts` class does, so the conversion may not be accurate.

Value

A `ts` object with the same data as `x`, and with starting time and frequency given by:

```
start = c(year(xstart), cycle(xstart))
frequency = frequency(x)
```

Note

The `tis` class covers more frequencies than the `ts` class does, so the conversion may not be accurate.

Author(s)

Jeff Hallman

See Also

as.ts

---

availablePort *Find Available TCP Port or Ports In Use*

Description

portsInUse returns a vector of UDP and TCP port numbers currently being used by the operating system. Under Linux, these are the ports listed in /proc/net/[u|t]cp, while for Windows they are the ports listed by netstat -an.

Under either operating system, availablePort returns the first TCP port number greater than 40000 that is not currenly in use.

Usage

```
portsInUse()
availablePort()
```

Value

portsInUse returns a numeric vector. availablePort returns an integer.

Author(s)

Jeff Hallman

See Also

hostName

badClassStop *Stop Function Execution on Wrong Class*

Description

Calls `stop` if `x` is not a `class`

Usage

```
badClassStop(x, class)
```

Arguments

`x`	an R object
`class`	character string

Author(s)

Jeff Hallman

basis *FAME time series attributes*

Description

FAME time series have (sometimes implicit) `basis` and `observed` attributes.

Usage

```
basis(x)
basis(x) <- value
observed(x)
observed(x) <- value
```

Arguments

`x`	a `tis` series
`value`	a character string, see the details

Details

A series `basis` is "business" or "daily", indicating whether the data values in a series are associated with a 5-day business week or a 7-day calendar week.

The `observed` attribute of series is one of the following:

`annualized`	Specifies that each time series value is the annualized sum of observations made throughout the associated ti
`averaged`	Specifies that each time series value is the average of the observations made throughout the associated time i
`beginning`	Specifies that each time series value represents a single observation made at the beginning of the associated t
`end`	Specifies that each time series value represents a single observation made at the end of the associated time int
`formula`	Specifies that the time series represents a transformation of other series. For time scale conversion and totalin
`high`	Specifies that each time series value is the maximum value for the time interval.
`low`	Specifies that each time series value is the minimum value for the time interval.
`summed`	Specifies that each time series value is the sum of observations made throughout the associated time interval.

Value

`basis` and `observed` return a character string. The assignment forms invisibly return `x`.

Author(s)

Jeff Hallman

References

The FAME documentation.

See Also

`getfame`, `putfame`

`between` *Check for Inclusion in a Closed Interval*

Description

Returns a logical vector like `y` showing if each element lies in the closed interval `[min(x1, x2), max(x1, x2)]`.

Usage

```
between(y, x1, x2)
```

Arguments

`y`	a numeric object
`x1`	a number
`x2`	a number

Value

A logical object like `y`.

Author(s)

Jeff Hallman

Examples

```
mat <- matrix(rnorm(16), 4, 4)
mat
between(mat, -2, 1)
```

`blanks` *Blanks*

Description

Takes an integer argument `n` and return a string of n blanks

Usage

```
blanks(n)
```

Arguments

`n`	an integer

Author(s)

Jeff Hallman

`cbind.tis` *Combine Series Into a Multivariate (Matrix) Time Indexed Series*

Description

This is `cbind` for `tis` objects. It binds several `ts` and `tis` objects together into a single matrix time indexed series.

Usage

```
## S3 method for class 'tis':
cbind(..., union = F)
```

Arguments

...	any number of univariate or multivariate ts or tis objects. All will be converted to tis objects by as.tis, and the result series all must have the same tif (time index frequency).
union	a logical. If union = F, a matrix created by the intersection of the time windows for the arguments will be created. If union = T, the union of the time windows will be used to create the matrix.

Details

If union is TRUE and the series in ... do not all start and end on the same time index, the missing observations are filled with NA.

The column names of the returned series are determined as follows:

If an argument was given a name in the call, the corresponding column has that name. If the argument was itself a matrix with column names, those will be used, otherwise the argument's name is expanded with digits denoting its respective columns.

Value

a multivariate tis object.

Note

Class "ts" has it's own cbind method which knows nothing about tis objects. R generic functions like cbind dispatch on the class of their first argument, so if you want to combine tis and ts objects by calling the generic cbind, be sure that the first argument is a tis, not a ts. You can always ensure this is the case by wrapping the first argument in ... in as.tis().

This function is a replacement for the FRB Splus3 function tdmatrix.

Author(s)

Jeff Hallman

See Also

cbind

Examples

columns *Rows and Columns of a Matrix*

Description

Create lists from the rows and/or columns of a matrix.

Usage

```
columns(z)
rows(z)
```

Arguments

z	a matrix

Value

`rows` returns a list of the rows of `z`. If `z` has row names, those will also be the names of the returned list.

`columns` does the same, but for columns. Note that if `z` is some kind of time series, so too will be the elements of the returned list.

Author(s)

Jeff Hallman

commandLineString *Command Line Arguments*

Description

`commandLineString` returns whatever followed `--args` on the command line that started R.

Usage

```
commandLineString()
```

Author(s)

Jeff Hallman

See Also

`commandArgs`

Examples

```
## Not run:
if(length(.cmd <- commandLineString()) > 0)
    try(source(textConnection(.cmd), echo = T, prompt.echo = "> "))
## End(Not run)
```

`constantGrowthSeries`

Constant Growth Series

Description

Create `tis` time series that grow at constant rates.

Usage

```
fanSeries(startValue, start, end, rates)
tunnelSeries(startValue, start, end, rate, spreads)
```

Arguments

`startValue`	starting value for the series at time `start`
`start`	a `ti` (Time Index) for the first observation.
`end`	a `ti` or something that can be turned into a `ti` giving the time index for the last observation.
`rates`	annual growth rate(s) for the series to be created
`rate`	annual growth rate for the series to be created
`spreads`	vector of 2 numbers giving the percentage values by which the starting values of the 'tunnel' series should be offset from `startValue`

Value

`fanSeries` returns a multivariate series that starts on `start` and ends on `end`. There are `length(rates)` columns. Each column begins at `startValue` and grows at the rate given by its corresponding element in `rates`. These are not true growth rates, rather each column has a constant first difference such that over the course of the first year, column i will grow `rates[i]` percent. This yields series that plot as straight lines.

`tunnelSeries` first calls `fanSeries` to create a univariate series running from `start` to `end` with a starting value of `startValue` and growing `rate` percent over the first year. It returns a bivariate series with columns that are offset from that series by `spreads[1]` and `spreads[2]` percent of the `startValue`.

Author(s)

Jeff Hallman

See Also

growth.rate

---

convert *Time scale conversions for time series*

Description

Convert tis series from one frequency to another using a variety of algorithms.

Usage

```
convert(x, tif, method = "constant", observed. = observed(x),
        basis. = basis(x), ignore = F)
```

Arguments

x	a univariate or multivariate tis series. Missing values (NAs) are ignored.
tif	a number or a string indicating the desired ti frequency of the return series. See help(ti)) for details.
method	method by which the conversion is done: one of "discrete", "constant", "linear", or "cubic".
observed.	"observed" attribute of the input series: one of "beginning", "end", "summed", "annualized", or "averaged". If this argument is not supplied and observed(x) != NULL it will be used. The output series will also have this "observed" attribute.
basis.	"daily" or "business". If this argument is not supplied and basis(x) != NULL it will be used. The output series will also have this "basis" attribute.
ignore	governs how missing (partial period) values at the beginning and/or end of the series are handled. For method == "discrete" or "constant" and ignore == T, input values that cover only part the first and/or last output time intervals will still result in output values for those intervals. This can be problematic, especially for observed == "summed", as it can lead to atypical values for the first and/or last periods of the output series.

Details

This function is a close imitation of the way FAME handles time scale conversions. See the chapter on "Time Scale Conversion" in the Users Guide to Fame if the explanation given here is not detailed enough.

Start with some definitions. Combining values of a higher frequency input series to create a lower frequency output series is known as aggregation. Doing the opposite is known as disaggregation.

Disaggration for "discrete" series: (i) for observed == "beginning" ("end"), the first (last) output period that begins (ends) in a particular input period is assigned the value of that input period. All other output periods that begin (end) in that input period are NA. (ii) for observed == "summed"

or "averaged", all output periods that end in a particular input period are assigned the same value. For "summed", that value is the input period value divided by the number of output periods that end in the input period, while for an "averaged" series, the output period values are the same as the corresponding input period values.

Aggregation for "discrete" series: (i) for observed == "beginning" ("end"), the output period is assigned the value of the first (last) input period that begins (ends) in the output period. (ii) for observed == "summed" ("averaged"), the output value is the sum (average) of all the input values for periods that end in the output period.

Methods "constant", "linear", and "cubic" all work by constructing a continuous function F(t) and then reading off the appropriate point-in-time values if observed == "beginning" or "end", or by integrating F(t) over the output intervals when observed == "summed", or by integrating F(t) over the output intervals and dividing by the lengths of those intervals when observed == "averaged". The unit of time itself is given by the `basis` argument.

The form of F(t) is determined by the conversion method. For "constant" conversions, F(t) is a step function with jumps at the boundaries of the input periods. If the first and/or last input periods only partly cover an output period, F is linearly extended to cover the first and last output periods as well. The heights of the steps are set such that F(t) aggregates over the input periods to the original input series.

For "linear" ("cubic") conversions, F(t) is a linear (cubic) spline. The x-coordinates of the spline knots are the beginnings or ends of the input periods if observed == "beginning" or "end", else they are the centers of the input periods. The y-coordinates of the splines are chosen such that aggregating the resulting F(t) over the input periods yields the original input series.

For "constant" conversions, if `ignore` == F, the first (last) output period is the first (last) one for which complete input data is available. For observed == "beginning", for example, this means that data for the first input period that begins in the first output period is available, while for observed == "summed", this means that the first output period is completely contained within the available input periods. If `ignore` == T, data for only a single input period is sufficient to create an output period value. For example, if converting weekly data to monthly data, and the last observation is June 14, the output series will end in June if `ignore` == T, or May if it is F.

Unlike the "constant" method, the domain of F(t) for "linear" and "cubic" conversions is NOT extended beyond the input periods, even if the ignore option is T. The first (last) output period is therefore the first (last) one that is completely covered by input periods.

Series with observed == "annualized" are handled the same as observed == "averaged".

Value

a `tis` time series covering approximately the same time span as `x`, but with the frequency specified by `tif`.

BUGS

Method "cubic" is not currently implemented for observed "summed", "annualized", and "averaged".

Author(s)

Jeff Hallman

References

Users Guide to Fame

See Also

aggregate, tif, ti

Examples

```
wSeries <- tis(1:105, start = ti(19950107, tif = "wsaturday"))
observed(wSeries) <- "ending"   ## end of week values
mDiscrete <- convert(wSeries, "monthly", method = "discrete")
mConstant <- convert(wSeries, "monthly", method = "constant")
mLinear   <- convert(wSeries, "monthly", method = "linear")
mCubic    <- convert(wSeries, "monthly", method = "cubic")

## linear and cubic are identical because wSeries is a pure linear trend
cbind(mDiscrete, mConstant, mLinear, mCubic)

observed(wSeries) <- "averaged"   ## weekly averages
mDiscrete <- convert(wSeries, "monthly", method = "discrete")
mConstant <- convert(wSeries, "monthly", method = "constant")
mLinear   <- convert(wSeries, "monthly", method = "linear")

cbind(mDiscrete, mConstant, mLinear)
```

---

csv *Writes a CSV (comma separated values) file.*

Description

Write a matrix or Time Indexed Series to a .csv file that can be imported into a spreadsheet.

Usage

```
csv(z, file = "", noDates = F, row.names = !is.tis(z), ...)
```

Arguments

z	matrix or tis object
file	either a character string naming a file or a connection. If "", a file name is constructed by deparsing z. The extension ".csv" is appended to the file name if it is not already there.
noDates	logical. If FALSE (the default) and z is a tis object, the first column of the output file will contain spreadsheet dates.

row.names	either a logical value indicating whether the row names of z are to be written along with z, or a character vector of row names to be written. If FALSE (the default) and z is a tis object, the first column of the output file will contain spreadsheet dates.
...	other arguments passed on to write.table.

Details

csv is essentially a convenient way to call write.table. If file is not a connection, a file name with the ".csv" extension is constructed. Next, a column of spreadsheet dates is prepended to z if necessary, and then csv calls

```
write.table(z, file = filename, sep = ",", row.names = !is.tis(z),
...)
```

Value

csv returns whatever the call to write.table returned.

Author(s)

Jeff Hallman

See Also

write.table

---

currentMonday *Day of Week Time Indexes*

Description

Return daily ti's for particular days of the week

Usage

```
currentMonday(xTi = today())
currentTuesday(xTi = today())
currentWednesday(xTi = today())
currentThursday(xTi = today())
currentFriday(xTi = today())
currentSaturday(xTi = today())
currentSunday(xTi = today())
latestMonday(xTi = today())
latestTuesday(xTi = today())
latestWednesday(xTi = today())
latestThursday(xTi = today())
latestFriday(xTi = today())
latestSaturday(xTi = today())
latestSunday(xTi = today())
```

Arguments

xTi	a ti object or something that the ti() function can turn into a ti object

Value

currentMonday returns the daily ti for the last day of the Monday-ending week that its argument falls into. currentTuesday returns the daily ti for the last day of the Tuesday-ending week that its argument falls into, and so on for the other weekdays.

latestMonday returns the daily ti for the last day of the most recent completed Monday-ending week that its argument falls into. Ditto for the other days of the week.

Author(s)

Jeff Hallman

See Also

ti

---

currentPeriod *Current Period Time Indexes*

Description

Return a current ti of the desired frequency

Usage

```
currentWeek(xTi = today())
currentMonth(xTi = today())
currentQuarter(xTi = today())
currentHalf(xTi = today())
currentYear(xTi = today())
currentQ4(xTi = today())
currentQMonth(xTi = today())
currentJanuary(xTi = today())
currentFebruary(xTi = today())
currentMarch(xTi = today())
currentApril(xTi = today())
currentMay(xTi = today())
currentJune(xTi = today())
currentJuly(xTi = today())
currentAugust(xTi = today())
currentSeptember(xTi = today())
currentOctober(xTi = today())
currentNovember(xTi = today())
currentDecember(xTi = today())
```

Arguments

xTi	a ti object or something that the ti() function can turn into a ti object

Details

currentWeek returns the weekly ti for the week that its argument falls into. If the argument is itself a ti, the returned week contains the last day of the argument's period. The default weekly frequency is "wmonday" (Monday-ending weeks), so currentWeek always returns wmonday ti's. This can be changed via the setDefaultFrequencies function.

All of the other current{SomeFreq} functions work the same way, returning the ti's of tif SomeFreq that the last day of their arguments period falls into. The tif's for currentHalf and currentQ4 are "semiannual" and "quarterly", respectively. Finally, currentQMonth returns the quarter-ending month of the currentQuarter of its argument.

currentJanuary returns the monthly ti for January of the January-ending year that the last day of its argument falls into. currentFebruary returns the monthly ti for February of the February-ending year that the last day of its argument falls into, and so on.

Value

All return return ti objects as described in the details.

Author(s)

Jeff Hallman

See Also

ti, tif, latestWeek setDefaultFrequencies

---

dateRange *Start and End Time Indices for a Series*

Description

Returns the starting and ending times of a series in a ti object of length 2.

Usage

```
dateRange(x)
```

Arguments

x	a ts or tis time series

Value

a ti (Time Index) object of length two. The first element is the starting time index, while the second is the ending time index.

Author(s)

Jeff Hallman

See Also

start, end, ti, tis

Examples

```
aTs <- ts(1:24, start = c(2001, 1), freq = 12)
aTis <- as.tis(aTs)
dateRange(aTs)
dateRange(aTis)
```

---

dayOfPeriod *Day positions in Time Index Periods*

Description

Return position within a ti period, or a particular day within the period.

Usage

```
dayOfPeriod(xTi = today(), tif = NULL)
dayOfWeek(xTi = today())
dayOfMonth(xTi = today())
dayOfYear(xTi = today())
firstDayOf(xTi)
lastDayOf(xTi)
firstBusinessDayOf(xTi)
lastBusinessDayOf(xTi)
firstBusinessDayOfMonth(xTi)
lastBusinessDayOfMonth(xTi)
currentMonthDay(xTi, daynum)
latestMonthDay(xTi, daynum)
```

Arguments

xTi	a ti object or something that the ti() function can turn into a ti object
tif	a time index frequency code or name. See tif.
daynum	day number in month

Details

The dayOfXXXXX functions all work the same way, returning the day number of the XXXXX that jul(xTi) falls on. For example, if today is Thursday, January 5, 2006, then dayOfWeek(), dayOfMonth() and dayOfYear() are all 5. All of these are implemented via dayOfPeriod, which converts its first argument to a Julian date (via jul(xTi)) and finds the ti with frequency tif that day falls into. It returns the day number of the period represented by that time index that the Julian date falls on.

firstDayOf and lastDayOf return a daily ti for the first or last day of the period represented by xTi. firstBusinessDayOf and lastBusinessDayOf do the same but the returned ti has business daily frequency.

firstBusinessDayOfMonth returns a business daily ti for the first business day of the month of xTi. lastBusinessDayOfMonth does the same but for the last business day of the month of xTi.

currentMonthDay returns a daily ti for the next upcoming daynum'th of the month. latestMonthDay does the same for the most recent daynum'th of the month.

currentMonday returns the daily ti for the last day of the Monday-ending week that its argument falls into. The other current{Weekday} functions work the same way.

Value

All of the functions except the dayOfXXXXX return ti objects as described in the details section above. The dayOfXXXXX functions return numbers.

Note

None of these business-day functions take account of holidays, so firstBusinessDayOfMonth(20010101), for example, returns January 1, 2001 which was actually a holiday. To see how to handle holidays, look at the holidays and nextBusinessDay help pages.

Author(s)

Jeff Hallman

See Also

ti, tif, jul, holidays, nextBusinessDay, previousBusinessDay

---

description *Description and Documentation Attributes*

Description

Get or set the description and documentation strings for an object.

Usage

```
description(x)
description(x) <- value
documentation(x)
documentation(x) <- value
```

Arguments

x	object whose description or documentation attribute is to be set or retrieved
value	a string

Value

The setters invisibly return x, the getters return the desired attribute or NULL.

Author(s)

Jeff Hallman

---

filter *Linear Filtering on a Time Series*

Description

Applies linear filtering to a univariate time series or to each series separately of a multivariate time series.

Usage

```
filter(x, ...)
## S3 method for class 'tis':
filter(x, ...)
## Default S3 method:
filter(x, filter, method = c("convolution", "recursive"),
       sides = 2, circular = FALSE, init, ...)
```

Arguments

x	a univariate or multivariate time series.
...	arguments passed along to filter.default, which is actually the version of filter from the stats package. The ... argument is ignored in filter.default
filter	a vector of filter coefficients in reverse time order (as for AR or MA coefficients).
method	Either "convolution" or "recursive" (and can be abbreviated). If "convolution" a moving average is used: if "recursive" an autoregression is used.

sides	for convolution filters only. If sides=1 the filter coefficients are for past values only; if sides=2 they are centred around lag 0. In this case the length of the filter should be odd, but if it is even, more of the filter is forward in time than backward.
circular	for convolution filters only. If TRUE, wrap the filter around the ends of the series, otherwise assume external values are missing (NA).
init	for recursive filters only. Specifies the initial values of the time series just prior to the start value, in reverse time order. The default is a set of zeros.

Details

Missing values are allowed in x but not in filter (where they would lead to missing values everywhere in the output).

Note that there is an implied coefficient 1 at lag 0 in the recursive filter, which gives

$$y_i = x_i + f_1 y_{i-1} + \cdots + f_p y_{i-p}$$

No check is made to see if recursive filter is invertible: the output may diverge if it is not.

The convolution filter is

$$y_i = f_1 x_{i+o} + \cdots + f_p x_{i+o-p-1}$$

where o is the offset: see sides for how it is determined.

Value

A tis time indexed series if x has class tis, otherwise a class ts time series.

Note

convolve(, type="filter") uses the FFT for computations and so *may* be faster for long filters on univariate series, but it does not return a time series (and so the time alignment is unclear), nor does it handle missing values. filter is faster for a filter of length 100 on a series of length 1000, for example.

See Also

convolve, arima.sim

Examples

```
x <- tis(1:100, start = c(2000,1), freq = 12)
filter(x, rep(1, 3))
filter(x, rep(1, 3), sides = 1)
filter(x, rep(1, 3), sides = 1, circular = TRUE)

filter(presidents, rep(1,3))
```

format.ti *Time Index Conversion to Character*

Description

Format a time index object for printing

Usage

```
format.ti(x, ..., tz = "GMT")
```

Arguments

x	a time index object
tz	A timezone specification to be used for the conversion if x has an intraday frequency. System-specific, but "" is the current time zone, and "GMT" is UTC.
...	other args passed on to format.POSIXlt.

Details

x is converted to a POSIXlt object and then format.POSIXlt takes over.

Value

a character vector representing x

Note

format.POSIXlt has been modified to understand two additional format symbols in addition to those documented in link{strftime}: "%q" in the format string is replaced with the quarter number (1 thru 4) and "%N" is replaced with the first letter of the month name.

Author(s)

Jeff Hallman

See Also

format.POSIXlt, strftime

Examples

```
format(today() + 0:9, "%x")
```

getfame *Fame Interface*

Description

getfame and putfame read and write time indexed series from and to Fame databases.

fameWhats returns information about an object in a database, including its name, class, type, basis and observed attributes, as well as start (a ti Time Index) and length. If getDoc is TRUE, it will also include description and documentation components. fameWhats is a wrapper around the function fameWhat, which provides the same information in a lower-level form.

fameWildlist returns a list giving the name, class, type and frequency of the objects in the database with names that match wildString.

Usage

```
getfame(sernames, db, save = F, envir = parent.frame(),
        start = NULL, end = NULL, getDoc = T)
putfame(serlist, db, access = "shared", update = T,
        checkBasisAndObserved = F, envir = parent.frame())
fameWhats(db, fname, getDoc = T)
fameWildlist(db, wildString = "?", nMax = 1000, charMode = T)
```

Arguments

sernames	character vector of Fame names of series and/or scalars to retrieve.
db	string giving the name of Fame database to read or write from. Full path names should not be used for registered databases.
save	if T the retrieved series are individually saved in the environment specified by envir.
envir	for getfame, the environment used by assign to save the retrieved series if save is T. For putfame, if serlist is a character vector, the environment in which to find the series that will be stored in the database. The default environment for both functions is the frame of the caller.
start	a ti object, or something that the ti function can turn into a ti object. The time index for the first observation in the returned series. The default is the series start in the database.
end	a ti object, or something that the ti function can turn into a ti object. The time index for the last observation in the returned series. The default is the series end in the database.
getDoc	if TRUE (the default), also get the series description and documentation attributes, accessible via functions of the same names.
serlist	the tis objects to be written to the database. This can either be a character vector giving the names of the series in the environment specified by envir, or it can be a list containing the series themselves.

access	string specifying the access mode to open the database in.
update	if TRUE (the default), existing series in the database will be updated. If FALSE, existing series in the database with the same names will be replaced by the series in serlist.
checkBasisAndObserved	if TRUE and update == TRUE, the basis and observed attributes of any existing series with the same name will be checked for consistency with the updating series from serlist. If the basis or observed attributes differ, the update will not happen.
fname	name of an object in a FAME database
wildString	string containing FAME wildcards
nMax	maximum number of matches to return
charMode	if TRUE (the default) return class, type and freq components as strings, rather than integer codes.

Details

Fame names vs. R names:

The R names of series may differ from their Fame names. For getfame, names(sernames) holds the R names of the retrieved series. If sernames does not have a names attributes, the R names will be the same as the Fame names.

Naming for putfame is more complicated, because the series specified by serlist for putfame may be univariate or multivariate. For a multivariate series, the column names of the matrix become the Fame names. Not having a name for each column is thus an error.

A univariate series may be a single-column matrix. If it is, and it has a column name, that becomes the Fame name of the series. Otherwise, the Fame name of a univariate series is the corresponding element of names(serlist). If serlist is an actual list of series, names(serlist) must be of the same length. For character vector serlist a names attribute is optional. If there isn't one, the Fame names will be the same as the R names.

Consistency checking when update == TRUE:

If there is already an existing series in the database with the same name as one in serlist, the Fame class, type, and frequency are checked for consistency between the updating series and the existing series in the database. In addition, if checkBasisAndObserved is TRUE, those attributes are also checked. Inconsistencies for any of the checked attributes between the updating existing series will abort the update. The default value for checkBasisAndObserved is set to FALSE because this inconsistency is very common in MRA code.

Value

getfame returns a list of the retrieved series. If save is T, the list is returned invisibly. The names of the list are the R names described in the details. Fame scalars are returned as strings created by the Fame type command. If getDoc is TRUE (the default), the retrieved series will also have attributes named description and documentation.

putfame invisibly returns an empty string.

Note

The Linux versions of these functions use the Fame HLI and a child server process. The function `fameRunning` is called to see if the server process is already running. If not, `fameStart` starts it and the HLI. Your .Last function should call `fameStop` to shut them down gracefully. In any given R session, once the Fame HLI has died for any reason, it cannot be restarted. This is a Fame limitation. On exit, `getfame` always closes whatever databases it opened, so there's no reason not to just leave the server running as long as the R session is alive. Death of the R process kills the server process as well.

Author(s)

Jeff Hallman

See Also

`ti`, `tis`, `startRemoteServer`

Examples

```
## Not run:
usdb <- "/fame/data/database/us.db"
boink <- getfame("gdp.q", db = usdb)     ## returns a list
gpd.q <- boink[[1]]                      ## or boink$gdp.q
getfame("gdp.q", db = usdb, save = TRUE) ## saves gdp.q in the current frame

## saves the series as "nominalIncome"
getfame(c(nominalIncome = "gdp.q"), db = usdb, save = TRUE)
## End(Not run)
seriesA <- tis(1:24, start = c(2002, 1), freq = 12)
seriesB <- tis(1:104, start = c(2002, 1), tif = "wmonday")
documentation(seriesB) <- paste("Line", 1:4, "of seriesB documentation")
## store them as "mser" and "wser"
putfame(c(mser = "seriesA", wser = "seriesB"), db = "myfame.db")

matrixSeries <- cbind(a = seriesA, b = seriesA + 3)
putfame(matrixSeries, db = "myfame.db") ## stores as "a" and "b" in Fame

fameWildlist("myfame.db")
fameWhats("myfame.db", fname = "wser", getDoc = TRUE)
```

`growth.rate` *Growth Rates of Time Series*

Description

Get or set growth rates of a `tis` time series in annual percent terms.

Usage

```
growth.rate(x, lag = 1, simple = T)
growth.rate(x, start = end(x) + 1, simple = T) <- value
```

Arguments

x	a tis time series or something that can be turned into one by as.tis
lag	number of lags to use in calculating the growth rate as outlined in the details below
simple	simple growth rates if TRUE, compound growth rates if FALSE
start	the first ti time index for which values of x should be replaced to make growth.rate(x[start]) == value[1].
value	desired growth rates

Details

An example: Suppose x is a quarterly series, then if simple is TRUE,

```
growth.rate(x, lag = 3) == 100 * ((x[t]/x[t-3]) - 1) * (4/3)
```

while if simple is FALSE

```
growth.rate(x, lag = 3) == 100 * ((x[t]/x[t-3])^(4/3) - 1).
```

Value

growth.rate(x) returns a tis series of growth rates in annual percentage terms.

Beginning with the observation indexed by start,

```
growth.rate(x) <- value
```

sets the values of x such that the growth rates in annual percentage terms will be equal to value. x is extended if necessary. The modified x is invisibly returned.

Author(s)

Jeff Hallman

---

hexidecimal *Hexidecimal conversions*

Description

Convert numeric vectors to hexidecimal strings and vice versa.

Usage

```
hexidecimal(dec)
hex2numeric(hex)
```

Arguments

dec	numeric vector
hex	character vector of hexidecimal strings

Details

Hexidecimals are base 16 numbers with digits represented by the characters '0123456789abcdef'. `hex2numeric("1df")`, for example, is 479 (256 + 13*16 + 15).

Hex numbers are often used to represent bits in a byte, since '9F' takes up less space than '10011111'. TCP/IP port numbers are also often represented in hex.

Value

`hexidecimal` returns a character object like `dec`.

`hex2numeric` returns a numeric object like `hex`.

Author(s)

Jeff Hallman

---

hms *Hours, Minutes and Seconds from a Time Index or Jul*

Description

Extract the fractional part of a `ti` (time index) or `jul` (julian date) object as a normalized list of hours, minutes, and seconds.

Usage

```
hms(x)
```

Arguments

x	a `jul` or something numeric that can be converted into a `jul` with a fractional part.

Details

The fractional part of `x` is multiplied by 86400 (the number of seconds in a day) and rounded to get the number of seconds. This is then divided by 3600 to get the number of hours, and the remainder of that is divided by 60 to get the normalized number of minutes. The remainder from the second division is the normalized number of seconds.

Value

A list with components:

hours	Normalized number of hours
minutes	Normalized number of minutes
seconds	Normalized number of seconds

See the details.

Note

Support for fractional days in ti and jul objects is relatively new and untested. There is probably code lurking about that assumes the numeric parts of ti and jul objects are integers, or even code that may round them to make sure they are integers. The fractional parts of ti and jul objects may not survive encounters with such code.

Author(s)

Jeff Hallman

See Also

ti and jul. Also see hourly for information on intraday frequencies

Examples

```
hms(today() + 0.5)
hms(today())
hms(today() + 43201/86400)
```

---

holidays *Holidays*

Description

Functions that know about Federal and FRB (Federal Reserve Board) holidays.

Usage

```
nextBusinessDay(x, holidays = NULL, goodFriday = F, board = F)
previousBusinessDay(x, holidays = NULL, goodFriday = F, board = F)
isHoliday(x, goodFriday = F, board = F)
isGoodFriday(x)
isEaster(x)
holidays(years, goodFriday = F, board = F)
federalHolidays(years, board = F)
goodFriday(years)
easter(years)
holidaysBetween(startTi, endTi, goodFriday = F, board = F)
```

Arguments

x	a ti time index, or something that can be turned into one, such as a yyyymmdd number or a Date object.
holidays	a vector of holidays (in yyyymmdd form) to skip over, or NULL. In the latter case, the holidays function is used to determine days to skip over.
goodFriday	if TRUE, consider Good Friday as a holiday. Default is FALSE because Good Friday is not a federal holiday.
board	if TRUE, the Friday preceding a Saturday NewYears, Independence, Veterans or Christmas Day is considered a holiday.
years	numeric vector of 4 digit years
startTi	a daily ti time index, or something that can be turned into one
endTi	a daily ti time index, or something that can be turned into one

Details

Federal law defines 10 holidays. Four of them, NewYears, Independence, Veterans and Christmas, fall on the same date every year. The other six fall on particular days of the week and months (MLK, Presidents, Memorial, Labor, Columbus, and Thanksgiving).

If one of the four fixed-date holidays falls on a Sunday, the federal holiday is celebrated the next day (Monday). If it falls on a Saturday, the preceding day (Friday) is a holiday for the Federal Reserve Board, but not for the Reserve Banks and the banking system as a whole.

Value

nextBusinessDay and previousBusinessDay return "business" frequency ti objects.

isHoliday, isGoodFriday and isEaster return Boolean vectors as long as x.

easter and goodFriday return numeric vectors of yyyymmdd dates of the appropiate holidays for each year in the years argument.

federalHolidays returns a numeric vector of yyyymmdd dates for the federal holidays for each year in years. The names attribute of the returned vector contains the holiday names.

holidays returns a vector like federalHolidays does. The only difference between the two functions is that holidays has the option of including Good Fridays.

holidaysBetween returns a vector of yyyymmdd dates for holidays that fall within the time spanned by [startTi, endTi].

Note

The algorithm for finding Easter dates was found somewhere on the web (I don't remember where) and is unbelievably complex. It would probably be simpler to just celebrate the home opener of the Cleveland Indians instead.

Author(s)

Jeff Hallman

hostName *DNS Host name*

Description

On Linux, this is just `tolower(Sys.info()["nodename"])`. On Windows the node name does not include the DNS suffix, which has to be found by parsing the output of the `ipconfig` program.

Usage

```
hostName()
```

Value

The DNS host name as a string.

Author(s)

Jeff Hallman

See Also

`availablePort` to find a port number that is not in use.

interpNA *Interpolate missing values in a Time Indexed Series*

Description

Calls `approxfun` or `splinefun` to interpolate missing values in a `tis` object.

Usage

```
interpNA(x, method = "constant", useTimes = F, offset = 1, rule = 2, f = 0, ...)
```

Arguments

`x`	a `tis` time series
`method`	One of c("constant", "linear", "fmm", "natural", "periodic"). Methods "constant" and "linear" call `approxfun`; the others call `splinefun`.
`useTimes`	if `TRUE`, use `time(x, offset)` (the decimal times of `x`) as the 'x' part of the (x, y) pairs used for interpolation. If `FALSE` (the default), use `ti(x)` (the integer time indices of `x`) as the 'x' part of the (x, y) pairs.

offset	if useTimes is TRUE, a number in the range [0,1] telling where in the periods represented by ti(x) to get the points for the 'x' parts of the (x, y) pairs. See the help for jul for a more detailed explanation of this parameter.
rule	For methods "constant" and "linear": an integer describing how interpolation is to take place outside the interval [min(x), max(x)]. If rule is 1 then NAs are returned for such points and if it is 2, the value at the closest data extreme is used.
f	For method="constant" a number between 0 and 1 inclusive, indicating a compromise between left- and right-continuous step functions. If y0 and y1 are the values to the left and right of the point then the value is y0*(1-f)+y1*f so that f=0 is right-continuous and f=1 is left-continuous.
...	Other arguments passed along to approxfun for methods "constant" and "linear".

Details

Depending on the method specified, a call to either approxfun or splinefun is constructed with appropriate arguments and executed for each column of x. In the call to approxfun or splinefun, the time indices ti(x) (or the decimal times returned by time(x, offset), if useTimes is TRUE) serve as the 'x' argument and the column values as the 'y' argument.

Value

A tis object like x with NA values filled in by interpolated values.

Author(s)

Jeff Hallman

See Also

approxfun, splinefun, ti

Intraday *R support for FAME-style Intraday frequencies*

Description

create tif (TimeIndexFrequency) codes for hourly, minutely, and secondly ti's.

Usage

```
  hourly(n = 0)
minutely(n = 0)
secondly(n = 0)
```

Arguments

n	number of base periods to skip. That is, `hourly(2)` gives a `tif` code for a series observed every 2nd hour, while both `minutely()` and `minutely(1)` are for a series observed once per minute, `secondly(30)` means every 30 seconds, and so on.

Details

The current implementation has hourly(n) –> 2000 + n, minutely(n) –> 3000 + n, and secondly(n) –> 4000 + n. If `n` divides evenly into 3600 for `secondly(n)`, the return code will be the same as `hourly(n/3600)`. For `secondly(n)` and `minutely(n)`, if `n` divides evenly into 60, the return code will be as if `minutely(n/60)` or `hourly(n/60)` had been called, respectively.

For `hourly(n)`, `n` must evenly divide into 24 and be less than 24, i.e., `n` is one of 1, 2, 3, 4, 6, 8, 12. For `minutely(n)`, `n` must be an even divisor of 1440, and less than 720. For `secondly(n)`, `n` must divide evenly into 86400, and be no larger than 960.

Value

An integer `tif` code.

Author(s)

Jeff Hallman

See Also

`tif`

---

isIntradayTif *Check for Intraday Time Index Frequency*

Description

The intraday frequencies are `hourly(n)`, `minutely(n)` and `secondly(n)`, where `n` is an appropriate integer. Their numeric `tif` codes are between 2000 and 4900, and that is what is actually checked for.

Usage

```
isIntradayTif(tif)
```

Arguments

tif	a character vector of `tif` names (see `tifName`) or a numeric vector of `tif` codes (see `tif` to be checked

Value

A logical vector as long as the input indicating which elements are intraday Time Index frequencies.

Note

The function does not attempt to verify if the supplied `tif` is actually valid, intraday or not.

Author(s)

Jeff Hallman

See Also

`hourly`, `minutely`, `secondly`

Examples

```
isIntradayTif(hourly(6))
isIntradayTif(tif(today()))
isIntradayTif(minutely(30))
```

---

isLeapYear *Check Leap Year*

Description

Checks whether or not the elements of its input are leap years.

Usage

```
isLeapYear(y)
```

Arguments

`y`	numeric vector of years

Details

`y` is a leap year if it is evenly divisible by 4 *and* either it is not evenly divisible by 100 or it is evenly divisible by 400, i.e., `y%%4 == 0 & (y%%100 != 0 | y%%400 == 0)`.

Value

logical vector of same length as `y` indicating whether or not the given years are leap years.

Author(s)

Jeff Hallman

Examples

```
isLeapYear(c(1899:2004))
```

`jul` *Julian Date Objects*

Description

The function `jul` is used to create `jul` (julian date) objects, which are useful for date calculations.

`as.jul` and `is.jul` coerce an object to a julian date and test whether an object is a `jul`.

Usage

```
jul(x, ...)
## S3 method for class 'Date':
jul(x, ...)
## S3 method for class 'ti':
jul(x, offset = 1, ...)
## Default S3 method:
jul(x, ...)
as.jul(x)
is.jul(x)
```

Arguments

`x`	object to be tested (`is.jul`) or converted into a `jul` object. As described in the details below, the constructor function `jul` can deal with several different kinds of `x`.
`...`	other args to be passed to the method called by the generic function. `jul.default` may pass these args to `as.Date`.
`offset`	for `jul.ti`, a number in the range [0,1] telling where in the period represented by `x` to find the day. 0 returns the first day of the period, while the default value 1 returns the last day of the period. For example, if `x` has `tif` = "wmonday" so that `x` represents a week ending on Monday, than any `offset` in the range [0, 1/7] will return the Tuesday of that week, while `offset` in the range (1/7, 2/7] will return the Wednesday of that week, `offset` in the range (6/7, 1] will return the Monday that ends the week, and so on.

Details

The `jul`'s for any pair of valid dates differ by the number of days between them. R's `Date` class defines a Date as a number of days elapsed since January 1, 1970, but `jul` uses the encoding from the *Numerical Recipes* book, which has Jan 1, 1970 = 2440588, and the code for converting between ymd and `jul` representations is a straightforward port of the code from that tome. This also matches the MRA Splus and csh (shell script) julian date routines.

Adding an integer to, or subtracting an integer from a `jul` results in another `jul`, and one `jul` can be subtracted from another. Two `jul`'s can also be compared with the operators (`==, !=, <. >, <=, >=`).

The `jul` class implements methods for a number of generic functions, including `"["`, `as.Date`, `as.POSIXct`, `as.POSIXlt`, `c`, `format`, `max`, `min`, `print`, `rep`, `seq`, `ti`, `time`, `ymd`.

`jul` is a generic function with specialized methods to handle `Date` and `ti` objects.

The default method (`jul.default`) deals with character `x` by calling `as.Date` on it. Otherwise, it proceeds as follows:

If `x` is numeric, `isYmd` is used to see if it could be yyyymmdd date, then `isTime` is called to see if `x` could be a decimal time (a number between 1799 and 2200). If all else fails, `as.Date(x)` is called to attempt to create a `Date` object that can then be used to construct a `jul`.

Value

`is.jul` returns `TRUE` or `FALSE`.

`as.jul` coerces its argument to have class `jul`, without making any attempt to discern whether or not this is a sensible thing to do.

`jul` constructs a `jul` object like `x`.

`jul` with no arguments returns the `jul` for the current day.

Note

The Julian calendar adopted by the Roman Republic was not accurate with respect to the rotational position of the Earth around the sun. By 1582 it had drifted ten days off. To fix this, Pope Gregory XIII decreed that the day after October 4, 1582 would be October 15, and that thereafter, leap years would be omitted in years divisible by 100 but not divisible by 400. This modification became known as the Gregorian calendar. England and the colonies did not switch over until 1752, by which time the drift had worsened by another day, so that England had to skip over 11 days, rather than 10.

The algorithms used in `jul2ymd` and `ymd2jul` cut over at the end of October 1582.

Author(s)

Jeff Hallman

References

Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (1992). *Numerical Recipes: The Art of Scientific Computing (Second Edition)*. Cambridge University Press.

See Also

`jul`, `ymd`, `ti`, `as.Date`

Examples

```
dec31 <- jul(20041231)
jan30 <- jul("2005-1-30")
jan30 - dec31               ## 30
feb28 <- jan30 + 29
jul()                       ## current date
```

---

lags *Lag a Time Series*

Description

Compute a lagged version of a time series, shifting the time base forward by a given number of observations. `lag` creates a single lagged series, while `lags` can create several lags at once.

Usage

```
## Default S3 method:
lag(x, k = 1, ...)
## S3 method for class 'tis':
lag(x, k = 1, ...)
lags(x, lags, name = "")
```

Arguments

`x`	A vector or matrix or univariate or multivariate time series (including `tis` series)
`k`	The number of lags
`...`	further arguments to be passed to or from methods
`lags`	vector of lag numbers
`name`	string or a character vector of names to be used in constructing column names for the returned series

Details

Vector or matrix arguments 'x' are coerced to time series.

For `lags`, column names are constructed as follows: If `name` is supplied and has as many elements as `x` has columns, those names are used as the base column names. Otherwise the column names of `x` comprise the base column names, or if those don't exist, the first `ncols(x)` letters of the alphabet are used as base names. Each column of the returned series has a name consisting of the basename plus a suffix indicating the lag number for that column.

Value

Both functions return a time series (`ts` or `tis`) object. If the `lags` argument to the `lags` function argument has more than one element, the returned object will have a column for each lag, with `NA`'s filling in where appropriate.

Note

This is a local version of `lag` that reverses the meaning of the sign of `k`. The standard R version of `lag` says that a series lagged by a positive `k` starts *earlier*. The OPPOSITE is true for this FRB version, to maintain consistency with the common usage of 'first lag, second lag' and so on here at FRB.

Author(s)

Jeff Hallman

`latestPeriod` *Most Recent Period Time Indexes*

Description

Return a `ti` for the most recent period of the desired frequency.

Usage

```
latestWeek(xTi = today())
latestMonth(xTi = today())
latestQuarter(xTi = today())
latestHalf(xTi = today())
latestYear(xTi = today())
latestQ4(xTi = today())
latestJanuary(xTi = today())
latestFebruary(xTi = today())
latestMarch(xTi = today())
latestApril(xTi = today())
latestMay(xTi = today())
latestJune(xTi = today())
latestJuly(xTi = today())
latestAugust(xTi = today())
latestSeptember(xTi = today())
latestOctober(xTi = today())
latestNovember(xTi = today())
latestDecember(xTi = today())
```

Arguments

`xTi`	a `ti` object or something that the `ti()` function can turn into a `ti` object

Details

The `latest{whatever}` functions are the same as the corresponding `current{whatever}` functions, except that they return the most recent completed `ti` of the desired frequency. A period is considered to be completed on it last day. For example, if today is Thursday, then `latestWedweek()` returns the week that ended yesterday. Yesterday it would have returned the same week, but the day before that (Tuesday) it would have returned the "wwednesday" `ti` for the week that had ended six days before.

`latestWeek` returns the weekly `ti` for the most recently completed week as of `xTi`. If the `xTi` is itself a `ti`, the returned week is the most recently completed week as of the last day of `xTi`. (Note that the default weekly frequency is "wmonday" (Monday-ending weeks), so `latestWeek` always returns "wmonday" `ti`'s.) See `setDefaultFrequencies` to change this.

All of the other `latest{SomeFreq}` functions work the same way, returning the `ti`'s for the most recently completed SomeFreq as of the last day of `xTi`. The `tif`'s (frequencies) for `latestHalf` and `latestQ4` are "semiannual" and "quarterly", respectively.

`latestJanuary` returns the monthly `ti` for January of the most recently completed January-ending year that the last day of its argument falls into. `latestFebruary` returns the monthly `ti` for February of the most recently completed February-ending year that the last day of its argument falls into, and so on.

Value

All return return `ti` objects as described in the details.

Author(s)

Jeff Hallman

See Also

`ti`, `tif`, `currentWeek` `setDefaultFrequencies`

---

linearSplineIntegration

Linear Spline Integration

Description

`lintegrate` gives the values resulting from integrating a linear spline, while `ilspline` returns linear splines that integrate to given values.

Usage

```
lintegrate(x, y, xint, stepfun = F, rule = 0)
ilspline(xint, w)
```

Arguments

x	x coordinates of the linear spline F defined by (x, y)
y	y coordinates of the linear spline F defined by (x, y)
xint	x intervals, i.e. [x[1], x[2]] is the first interval, [x[2], x[3]] is the second interval, and so on.
stepfun	if TRUE, F is a left-continuous step function. The default (FALSE) says F is continuous.
rule	one of {0, 1, NA} to specify the behavior of F outside the range of x. Use zero if F is zero outside the range of x, NA if F is NA outside the range of x, and one if F is to be linearly extended outside the range of x.
w	values the linear spline must integrate to

Details

lintegrate integrates the linear spline F defined by (x,y) over the xint intervals. The value of F outside the range of x is specified by the rule argument:

```
rule == 0  --> F(z) = 0 for z outside the range of x
rule == NA --> F(z) = NA for z outside the range of x
rule == 1  --> F(z) extended for z outside the range of x
```

If stepfun is TRUE, F(z) is assumed to be a left-continuous step function and the last value of y is never accessed.

(x[i], y[i]) pairs with NA values in either x[i] or y[i] NA are ignored in constructing F.

ilspline finds linear splines that integrate over the N intervals specified by the monotonically increasing N+1 vector xint to the N values given in w. The function finds N-vectors x and y such that:

```
(i)  x[j] = (xint[j-1] + xint[j])/2, i.e., the values of x are the
     midpoints of the intervals specified by xint, and

(ii) the linear spline that passes through the (x[i], y[i]) pairs (and
     is extended to xint[1] and xint[N+1] by linear
     extrapolation) integrates over each interval [xint[j],xint[j+1]]
     to w[j].
```

In fact, w can actually be an M by N matrix, in which case the y found by the function is also an M by N matrix, with each column of y giving the y coordinates of a linear spline that integrates to the corresponding column of w.

Value

lintegrate returns a vector of length length(xint) - 1.

ilspline returns a list with components named 'x' and 'y'.

Author(s)

Jeff Hallman

References

put references to the literature/web site here

See Also

spline, approx

Examples

```
w <- 10 + cumsum(rnorm(10))
blah <- ilspline(1:11, w)
ww <- lintegrate(blah$x, blah$y, 1:11, rule = 1)
w - ww  ## should be all zeroes (or very close to zero)
```

---

lines.tis *Plotting Time Indexed Series*

Description

Plotting methods for tis objects

Usage

```
lines.tis(x, midPoints = T, offset = 1, dropNA = F, ...)
points.tis(x, midPoints = T, offset = 1, dropNA = F, ...)
```

Arguments

x	a tis (time indexed series) object
midPoints	if TRUE (the default) and if offset was not specified, then offset is set to 0.5.
offset	a number in the range [0,1] telling where in each period of x to find the day on which to plot the point. 0 means the first day of each period; the default value 1 plots on the last day of each period. For example, if x has tif = "wmonday" so that each observation represents a week ending on Monday, than any offset in the range [0, 1/7] will plot points on Tuesdays, while offset in the range (1/7, 2/7] will plot on Wednesdays, offset in the range (6/7, 1] will plot on Mondays, and so on.
dropNA	if TRUE, observations with NA values are dropped before calling lines.default or points.default. See the details for why you might or might not want to do this. The default is FALSE, to match the behavior of lines.default and points.default.
...	other arguments to be passed on to lines.default or points.default.

Details

These are fairly simple wrappers around the `lines.default` and `points.default`. For example, `lines.tis` basically does this:

lines.default(x = time(x, offset = offset), y = x, ...)

and `points.tis` is similar. If `dropNA` is `TRUE`, the observations in `x` that are `NA` are dropped from the x and y vectors sent to the .default functions. For `points`, this shouldn't matter, since `points.tis` omits points with `NA` values from the plot.

For `lines` the `dropNA` parameter does make a difference. The help document for `lines` says:

"The coordinates can contain `NA` values. If a point contains `NA` in either its `x` or `y` value, it is omitted from the plot, and lines are not drawn to or from such points. Thus missing values can be used to achieve breaks in lines."

Note that if the `type` is one of `c("p", "b", "o")`, the non-`NA` points are still drawn, but line segments from those points to adjacent `NA` points are not drawn. If `dropNA = TRUE`, the `NA` points are dropped before calling `lines.default`, and all of the remaining points will be connected with line segments (unless suppressed by the `type` argument).

Author(s)

Jeff Hallman

See Also

`lines`, `points`

---

mergeSeries *Merge Time Indexed Series*

Description

Merge two time-indexed series using either the levels or the first differences of the second series where the series overlap.

Usage

```
mergeSeries(x, y, differences = F)
```

Arguments

`x, y`	`tis` objects, or objects that can sensibly be coerced to `tis` by `as.tis`.
`differences`	if `T`, the first differences of series are merged, and then cumulatively summed. The default is `F`.

Details

`x` and `y` must have the same `tif` (ti frequency), and the same number of column (if they are multivariate).

Value

A tis object series with start and end dates that span those of x and y. Where the series overlap, values from y are used.

Author(s)

Jeff Hallman

See Also

cbind.tis

---

naWindow *Exclude NA Observations*

Description

Windows a tis series to cut off leading and trailing NA observations.

Usage

```
naWindow(x, union = F)
```

Arguments

x	a tis time indexed series
union	see details below

Details

For multivariate (multiple columns) series and union = TRUE, a row of x is considered to be NA if and only if all entries in that row are NA. If union = FALSE (the default), a row is considered to be NA if any of its entries is NA.

Value

A copy of x with leading and trailing NA observations deleted.

Author(s)

Jeff Hallman

See Also

window

`osUtilities` *Operating System Utilities*

Description

`user` returns the user id of the current user.

`groups` returns a character vector of the Linux security groups `user.` is a member of.

`pid`, `pgid` and `ppid` return the process, group and parent process ID numbers, respectively, of the current R process.

`killProcess` kills a specified process.

`pwd` returns the process working directory.

`runningLinux` is shorthand for `Sys.info()["sysname"] == "Linux"`.

`runningWindows` is shorthand for `.Platform$OS.type == "windows"`.

Usage

```
user()
groups(user. = user())
pid()
pgid()
ppid()
killProcess(pid)
pwd()
runningLinux()
runningWindows()
```

Arguments

`user.`	a user id
`pid`	a process id number

Author(s)

Jeff Hallman

`pad.string` *Pad strings to a particular length*

Description

Pads a collection of strings to a desired length to either the left or right with a specified character.

Usage

```
pad.string(x, len = max(nchar(x)), padchar = " ", right = T)
```

Arguments

`x`	character object
`len`	desired length of the returned strings
`padchar`	a single-character string.
`right`	if `TRUE` (the default), the strings in `x` are padded to the right, else they are padded to the left.

Value

an object like `x`, but with the strings padded out to the desired length. Strings in `x` that are already `len` or more characters long are unaffected by the function.

Author(s)

Jeff Hallman

See Also

`format`

Examples

```
pad.string(c("aaa", "bbbbbb", "ccccccccccc"), len = 7, padchar = "X")
```

print.tis *Printing Time Indexed Series*

Description

Print method for time indexed series.

Usage

```
print.tis(x, format = "%Y%m%d", matrix.format = F, ...)
```

Arguments

x	a time indexed series
format	a character string describing how to format the observation times if either x is printed in matrix form. Format strings are detailed in format.ti.
matrix.format	TRUE or FALSE. See details.
...	additional arguments that may be passed along to print.ts. See details.

Details

If matrix.format is F (the default) and x is a univariate monthly, quarterly or annual series, printing is accomplished by print(as.ts(x), ...). Otherwise, x is printed as a matrix with rownames created by formatting the time indexes of the observations according to the format string.

Author(s)

Jeff Hallman

See Also

format.ti, print.ts

Examples

```
print(tis(1:31, start = today() - 30), format = "%b %d, %Y")
```

rowMeans *Form Row Sums and Means*

Description

Form row sums and means for numeric arrays.

Usage

```
rowSums (x, ...)
rowMeans(x, ...)
## Default S3 method:
rowSums(x, na.rm = FALSE, dims = 1, ...)
## Default S3 method:
rowMeans(x, na.rm = FALSE, dims = 1, ...)
## S3 method for class 'tis':
rowSums(x, ...)
## S3 method for class 'tis':
rowMeans(x, ...)
```

Arguments

x	an array of two or more dimensions, containing numeric, complex, integer or logical values, or a numeric data frame, or a tis time indexed series
...	arguments passed along to rowSums.default or rowMeans.default, which are actually the versions of rowSums and rowMeans from the base package. The ... argument is ignored in rowSums.default and rowMeans.default.
na.rm	logical. Should missing values (including NaN) be omitted from the calculations?
dims	Which dimensions are regarded as "rows" or "columns" to sum over. For row*, the sum or mean is over dimensions dims+1, ...; for col* it is over dimensions 1:dims.

Details

These functions are equivalent to use of apply with FUN = mean or FUN = sum with appropriate margins, but are a lot faster. As they are written for speed, they blur over some of the subtleties of NaN and NA. If na.rm = FALSE and either NaN or NA appears in a sum, the result will be one of NaN or NA, but which might be platform-dependent.

Value

A numeric or complex array of suitable size, or a vector if the result is one-dimensional. The dimnames (or names for a vector result) are taken from the original array.

If there are no values in a range to be summed over (after removing missing values with na.rm = TRUE), that component of the output is set to 0 (rowSums) or NA (rowMeans), consistent with sum and mean.

The tis-specific methods also return a tis.

See Also

apply, rowsum, and colSums for more details and examples.

Examples

```
mat <- tis(matrix(1:36, ncol = 3), start = latestJanuary())
cbind(mat, rowSums(mat), rowMeans(mat))
```

---

sendSocketObject *Send and Receive* R *Objects through Sockets*

Description

These functions allow R sessions on different machines to send R objects to each other. Consider two R sessions running on (possibly) different machines. Call the sessions R1 and R2, with R1 running on somehost.somewhere.com. To send an object from R1 to R2, the sequence of events goes like this:

1. R1 invokes receiveSocketObject(NNNNN), where NNNNN is an available port number on R1's localhost. (Use availablePort to find a port number.) The R1 session hangs until:

2. R2 invokes sendSocketObject(object, "somehost.somewhere.com", NNNNN) to send object to R1. If R1 is not listening on port NNNNN already, this operation will hang R2 until R1 invokes receiveSocketObject(NNNNN) to start listening there.

3. R1 and R2 both return after object has been sent, with receiveSocketObject on R1 returning the object that was sent.

Usage

```
sendSocketObject(object, host, port)
receiveSocketObject(port)
```

Arguments

object	an R object
host	DNS name of the host to which object is being sent
port	TCP port number on the receiving end

Details

sendSocketObject opens a blocking binary client socket connection to the given host and port, serializes object onto the connection, then closes it.

receiveSocketObject listens on a blocking binary server socket and returns the R object sent over it.

Value

`receiveSocketObject()` returns the object sent.

Note

`serialize` puts `object` on the socketConnection, while codeunserialize reads it at the other end.

Author(s)

Jeff Hallman

See Also

`availablePort` to find a port number, and `startRemoteServer` to run a server that accepts R commands over a socket and returns the resulting R objects

---

setDefaultFrequencies

Return known Time Index Frequencies, Change Default Frequencies

---

Description

A `tif` (Time Index Frequency) can usually be set either by code (a number) or by name. `setDefaultFrequencies` sets particular frequencies for the `tif` names "weekly", "biweekly", "bimonthly" (also "bimonth"), "quarterly" (also "q"), "annual" (also "a"), and "semiannual" (also "sann").

`tifList` returns the map of frequency names to frequency codes.

Usage

```
setDefaultFrequencies(weekly     = "wmonday",
                      biweekly   = "bw2wednesday",
                      bimonthly  = "bimonthdecember",
                      quarterly  = "qdecember",
                      annual     = "anndecember",
                      semiannual = "sanndecember",
                      setup = FALSE)
tifList()
```

Arguments

`weekly`	A string giving the name of the particular frequency that frequency "weekly" will correspond to
`biweekly`	Ditto for "biweekly"
`bimonthly`	Ditto for "bimonth" and "bimonthly"
`quarterly`	Ditto for "q" and "quarterly"

annual	Ditto for "a" and "annual"
semiannual	Ditto for "sann" and "semiannual"
setup	If TRUE, set all of the defaults, otherwise set only the defaults for which arguments were given. The default is FALSE, but see the details

Details

The named vector .tifList (returned by the function of the same name) stored in the global enviroment contains the mapping of frequency names to frequency codes. Running this function modifies the tifList vector and stores it back in the global environment. It gets run with setup = TRUE when the fame package is loaded. If you want different defaults, call the function sometime after that.

Value

A copy of the .tifList vector.

Author(s)

Jeff Hallman

See Also

tifName

---

solve.tridiag *Solve a Tridiagonal System of Equations*

Description

This function solves the equation a %*% x = b for x, where a is tridiagonal and b can be either a vector or a matrix.

Usage

```
## S3 method for class 'tridiag':
solve(a, b, ...)
```

Arguments

a	a tridiag object: a square tridiagonal (all zeroes except for the main diagonal and the diagonals immediately above and below it) matrix containing the coefficients of the linear system.
b	a vector or matrix giving the right-hand side(s) of the linear system. If missing, b is taken to be an identity matrix and the function will return the inverse of a.
...	ignored

Details

Uses the LINPACK `dgtsv` routine.

See Also

`solve`

`ssDate` *ssDate Objects*

Description

The function `ssDate` is used to create `ssDate` (spreadsheet date) objects, which are useful for reading and writing dates in spreadsheet form, i.e., as the number of days since December 30, 1899.

`as.ssDate` and `is.ssDate` coerce an object to a ssDate and test whether an object is a `ssDate`.

Usage

```
ssDate(x, ...)
as.ssDate(x)
is.ssDate(x)
```

Arguments

`x`	object to be tested (`is.ssDate`) or converted into a `ssDate` object.
`...`	other args to be passed to `jul` function.

Details

an `ssDate` is essentially a rebased Julian date that represents a date as the number of days since December 30, 1899. The constructor function `ssDate` subtracts `jul(18991230)` from `jul(x, ...)` and coerces the result to class `ssDate`. Pretty much all of the stuff you can do with `jul` objects can also be done with `ssDate` objects.

Value

`is.ssDate` returns `TRUE` or `FALSE`.

`as.ssDate` coerces its argument to have class `ssDate`, without making any attempt to discern whether or not this is a sensible thing to do.

`ssDate` constructs a `ssDate` object like `x`.

`ssDate` with no arguments returns the `ssDate` for the current day.

Author(s)

Jeff Hallman

See Also

jul

Examples

```
dec31 <- ssDate(20041231)
jan30 <- ssDate("2005-1-30")
jan30 - dec31              ## 30
feb28 <- jan30 + 29
ssDate()                      ## current date
```

---

clientServerR *Client - Server* R

Description

These functions implement R client and server sessions.

The first function, `serveHostAndPort()` runs on the server end of the session. It finds an available port on the host it is running on, and uses that host and port as the server end of a `serverSession` with the given client host and port. It returns the `serverSession` to the client and begins a loop listening on the serverPort for R objects to be sent through.

When an object arrives on the port, what happens depends on its mode. If the new arrival is an expression or call, it gets evaluated and the result is sent back to the client. If the arriving object is a character string, it gets parsed and evaluated and the result sent back. Any other mode of arriving object is just sent back. All attempted evaluations take place in a `try` block, and a `try-error` gets returned to the client if the parsing or evaluating fails.

After handling the incoming object, the server returns to the top of the loop. There are two ways to break out of the loop. The first is to send the expresssion `break` to the server. The other way, timing out, works only if the server runs on a real operating system, i.e., Unix or Linux. If so, the server sets an alarm timer when first starting the loop, and resets it after each pass through the loop. If the timer expires due to inactivity, the SIGALRM signal is sent to the server R process, which kills it.

All of the other functions documented here run on the client end of the session.

`startRemoteServer()` assumes that the machine it is running on is to be the client end of an R server session. It uses `ssh` to invoke `serveHostAndPort()` on the remote host, and waits for the remote server to return a serverSession object, which it then stores in the global environment as ".serverSession".

`print.serverSession` is a print method for serverSession objects.

`serverSession()` returns the serverSession stored in the global environment.

`hasExpired()` checks a serverSessions timestamp, which is updated on each send or receive operation, against it's timeout and the current time. Note that there is no good way to check with the server itself, since it may not be running. If the server has already died, attempting to contact it will likely hang the current R session.

endServerSession() checks to see if there is a serverSession stored in the global environment, exiting if there isn't. Assuming there is a session and it has not yet expired, the break expression is sent to kill it. If the session timestamp indicates that the session has expired, the function asks the remote machine to kill the session serverPid process. If the process has already died, this does no harm.

validServerIsRunning() answers TRUE if serverSession() returns a serverSession that has not timed out yet.

ensureValidServer returns the current serverSession if there is one, or starts a new one and returns that.

sendToServer() and receiveFromServer do what their names imply.

Finally, sendExpression sends its unevaluated argument (retrieved via substitute) to the server, and waits for the server to return a result, which becomes the return value of sendExpression.

Usage

```
serveHostAndPort(clientHost, clientPort, timeout = 3600, quitAfter = T)
startRemoteServer(host = getOption("remoteHost"), user. = user(), timeout = 3600)
hasExpired(ss)
print.serverSession(x, ...)
serverSession()
endServerSession()
validServerIsRunning(fail = F)
ensureValidServer(...)
sendToServer(object)
receiveFromServer()
sendExpression(expr)
```

Arguments

clientHost	DNS name of the machine the R client is running on
clientPort	Port number on the client
timeout	number of seconds of inactivity after which the R server commits suicide
quitAfter	if TRUE (the default), upon breaking out of the receive-eval-return result loop, the server invokes q("no") to kill itself. For testing, you may want to start the remote server by hand and invode serveHostAndPort with quitAfter = FALSE to see what is happening on the far end.
host	DNS name of the machine the R server is to run on
user.	user name the R server is to run under
ss	a serverSession object
x	a serverSession object
...	For print.servserSession, arguments to be passed on to print.simple.list. For ensureServerSession, arguments to be passed through to startRemoteServer
fail	logical. If TRUE (not the default), raise an error exception if there is not a valid server session running
object	arbitrary R object to send to the server

`expr` an expression (entered without quotes) to send to the remote R server, where it will be evaluated and the result returned.

Value

`serverSession()` returns the stored serverSession, while `hasExpired()` and `validServerIsRunning()` return `TRUE` or `FALSE`.

`sendExpression()` returns the value that the given expression evaluated to on the server.

The other functions return nothing of interest.

Note

Getting the detailed sequencing right for blocking sockets can be very frustrating, since the R processes at both ends tend to hang on every mistake. The easiest way to get the sequence right is to just use `startRemoteServer()` from the client end to get everything started, and then use `sendExpression()` to send expressions to the server and get results on the client. Finally, you can call `endServerSession()` when you are finished to clean up, as in the example below. An even better idea is to leave the server running, but call `endServerSession()` in your `.Last` function.

The remote server is started via ssh, and this will usually require you to interactively supply a password. No facility to store your password and use it when needed has been provided, as that would just be asking for security trouble. On Windows, the `ssh` function uses the `plink` program to establish the ssh connection, and uses it's password prompt as well.

Author(s)

Jeff Hallman

See Also

`ssh`

Examples

```
## Not run:
startRemoteServer(host = "mralx2")
blah <- sendExpression(getfame("gdp.q", db = "us"))
endServerSession()
## End(Not run)
```

---

start.tis *Starting and ending time indexes*

Description

Return the start or end time index for a `tis` object.

Usage

```
## S3 method for class 'tis':
start(x, ...)
## S3 method for class 'tis':
end(x, ...)
start(x) <- value
```

Arguments

x	a tis object
value	desired start attribute
...	ignored

Value

start.tis returns the start attribute of x, while end.tis returns start(x) + nobs(x) - 1. start(x) <- value returns the series x shifted such that it's starting time is value.

Note

start and end are generic functions with default methods that assume x has (or can be given) a tsp attribute. The default methods return a two vector as c(year, period), while the methods described here return infinitely more useful ti objects.

Author(s)

Jeff Hallman

See Also

start,end

Examples

```
x <- tis(numeric(8), start = c(2001, 1), freq = 4)
start(x)           ## --> ti object representing 2001Q1
start(as.ts(x))    ## --> c(2001, 1)
```

---

stripBlanks *Strip Blanks*

Description

Strips leading and trailing blanks from strings

Usage

```
stripBlanks(strings)
```

Arguments

strings	character vector

Value

An object like strings with no leading or trailing blanks.

Author(s)

Jeff Hallman

See Also

blanks, gsub

---

stripClass *Remove part of a class attribute*

Description

An R object may have a class attribute that is a character vector giving the names of classes it inherits from. stripClass strips the class classString from that character vector. stripTis(x) is shorthand for stripClass(x, "tis").

Usage

```
stripClass(x, classString)
stripTis(x)
```

Arguments

x	an object whose class character vector may or may not include classString
classString	name of class to remove from the inheritance chain

Value

An object like x, but whose class attribute does not include classString. If the class attribute less classString is empty, unclass(x) is returned.

Note

This function can be useful in functions that return a modified version of one their arguments. For example, the format.ti method takes a ti (TimeIndex) as an argument and returns a character object object 'like' the original argument. The first thing format.ti(x) does internally is z <- stripClass(x, "ti"). This creates z as a copy of x but with the difference that z no longer inherits from class ti. The function then fills in the data elements of z with the approriate strings and returns it. The beauty of this approach is that the returned z already has all of the attributes x had, except that it no longer inherits from class ti. In particular, if x was a matrix with dimnames, etc., z will also have those attributes.

Author(s)

Jeff Hallman

See Also

class

`tiDaily` *Daily and Business Day Time Indexes*

Description

Return a daily or business day `ti` corresponding to a specified position within a time index.

Usage

```
tiDaily(xTi, offset = 1)
tiBusiness(xTi, offset = 1)
```

Arguments

`xTi`	a `ti` object or something that the `ti()` function can turn into a `ti` object
`offset`	for `ti xTi`, a number in the range [0,1] telling where in the period represented by `x` to find the day. 0 means the first day of the period, 1 the last day of the period, and fractional values for in-between day.

Value

`tiDaily` converts its first argument to a `jul` using the offset provided, and returns a daily `ti` for that day.

`tiBusiness` converts its first argument to a `jul` using the offset provided, and returns a "business" `ti` for that day.

Author(s)

Jeff Hallman

See Also

ti, jul

tif2freq *Periods Per Year for Time Index Frequencies*

Description

Returns the frequency of a `ti` object constructed from the current date with the given `tif`.

Usage

```
tif2freq(tif)
```

Arguments

`tif`	a `tifName` or `tif` code

Value

a number

Author(s)

Jeff Hallman

See Also

`tif`, `tifName`, `frequency`

Examples

```
tif2freq("wmonday")
tif2freq("monthly")
tif2freq(tif(today()))
```

tif *Time Index Frequencies*

Description

Return the tif code of an object, or the name associated with a tif code.

Usage

```
tif(x, ...)
## S3 method for class 'ti':
tif(x, ...)
## S3 method for class 'tis':
tif(x, ...)
## S3 method for class 'ts':
tif(x, ...)
## Default S3 method:
tif(x, freq = NULL, ...)
tifName(s)
## Default S3 method:
tifName(s)
## S3 method for class 'ti':
tifName(s)
## S3 method for class 'tis':
tifName(s)
```

Arguments

x	a ti or tis object, or a string giving a tif name.
freq	numeric. If x is missing, return the tif for this frequency, otherwise ignore.
...	ignored
s	a ti or tis object, or a tif code.

Details

The tifList object associates tifNames with tif codes. Most functions that call for tif argument can take either a tif code or a tif name.

Both function are generic function with methods for ti and tis objects, as well as a default method. tif also has a method for ts objects.

Value

tif returns the tif code for x, while tifName returns a name for that code. Many of the codes have several names, but only the default one is returned.

tif or tifName called with no arguments returns a vector of all tif codes with names.

Author(s)

Jeff Hallman

See Also

ti, frequency

Examples

```
tif()                  ## returns a vector of all tif codes
tifName(today())       ## today() returns a ti
```

---

tifToFameName *FAME Names for Time Index Frequencies*

Description

Returns the FAME names for the given time index frequencies.

Usage

```
tifToFameName(tif)
```

Arguments

tif	character vector of tifNames or a numeric vector of tif codes.

Value

A character vector as long as the input giving the FAME names of the input.

Author(s)

Jeff Hallman

See Also

tif, tifName

Examples

```
tifToFameName(tif(today()))
tifToFameName(tif(latestMonth()))
tifToFameName(tifName(today()))
tifToFameName(tifName(latestMonth()))
```

ti *Time Index Objects*

Description

The function ti is used to create time index objects, which are useful for date calculations and as indexes for tis (time indexed series).

as.ti and is.ti coerce an object to a time index and test whether an object is a time index.

couldBeTi tests whether or not x is numeric and has all elements within the range expected for a ti time index with the given tif. If tif is NULL (the default), the test is whether or not x could be a ti of *any* frequency. If so, it can be safely coerced to class ti by as.ti.

Usage

```
ti(x, ...)
ti.Date(x, ...)
ti.default(x, tif = NULL, freq = NULL, ...)
ti.jul(x, tif = NULL, freq = NULL, hour = 0, minute = 0, second = 0, ...)
ti.ssDate(x, ...)
ti.ti(x, tif = NULL, freq = NULL, ...)
ti.tis(x, ...)
as.ti(x)
is.ti(x)
couldBeTi(x, tif = NULL)
```

Arguments

x	object to be tested (is.ti) or converted into a ti object. As described in the details below, the constructor function ti can deal with several different kinds of x.
hour	used if and only if codetif is an intraday frequency
minute	used if and only if codetif is an intraday frequency
second	used if and only if codetif is an intraday frequency
...	other args to be passed to the method called by the generic function.
tif	a ti Frequency, given as either a numerical code or a string. tif() with no arguments returns a list of the allowable numerical codes and names. Either tif or freq must be supplied for the variants of ti().
freq	some tif's can alternatively be specified by their frequency, such as 1 (annual), 2 (semiannual), 4 (quarterly), 6 (bimonthly), 12 (monthly), 24 (semimonthly), 26 (biweekly), 36 (tenday), 52 (weekly), 262 (business) and 365 (daily). Either tif or freq must be supplied for the variants of ti().

Details

A ti has a tif (ti Frequency) and a period. The period represents the number of periods elapsed since the base period for that frequency. Adding or subtracting an integer to a ti gives another ti. Provided their corresponding element have matching tifs, the comparison operators <, >, <=, >=, == all work, and subtracting one ti from another gives the number of periods between them. See the examples section below.

The ti class implements methods for a number of generic functions, including "[", as.Date, as.POSIXct, as.POSIXlt, c, cycle, edit, format, frequency, jul, max, min, print, rep, seq, tif, tifName, time, ymd.

ti is a generic function with specialized methods to handle jul, Date, ti and tis objects.

The default method (ti.default) deals with character x by calling as.Date on it. Otherwise, it proceeds as follows:

If x is numeric, a check is made to see if x could be a ti object that has somehow lost it's class attribute. Failing that, isYmd is used to see if it could be yyyymmdd date, then isTime is called to see if x could be a decimal time (a number between 1799 and 2200). If x is of length 2, an attempt to interpret it as a c(year, period) pair is made. Finally, if all else fails, as.Date(x) is called to attempt to create a Date object that can then be used to construct a ti.

Value

is.ti and couldBeTi return TRUE or FALSE.

as.ti coerces its argument to have class ti, without making any attempt to discern whether or not this is a sensible thing to do. as.ti should only be called on an object if couldBeTi on that object answers TRUE.

ti constructs a ti object like x, except for two special cases:

1. If x is a tis series, the return value is a vector time index with elements corresponding to the observation periods of x.

2. If x is a numeric object of length 2 interpretable as c(year, period), the return value is a single ti.

Note

The as.Date(x) call is not wrapped in a try-block, so it may be at the top of the stack when ti fails.

Author(s)

Jeff Hallman

See Also

jul, ymd, tif, tifName, as.Date

Examples

```
z <- ti(19971231, "monthly")    ##  monthly ti for Dec 97
is.ti(z)                        ##  TRUE
is.ti(unclass(z))               ##  FALSE
couldBeTi(unclass(z))           ##  TRUE
ymd(z + 4)                      ##  19980430
z - ti(c(1997,6), freq = 12)    ##  monthly ti for June 1997
ti(z, tif = "wmonday")          ##  week ending Monday June 30, 1997
```

---

tisFromCsv *Read time series from Comma Separated Values (.csv) file*

Description

Reads `tis` (Time Indexed Series) from a csv file, returning the series in a list, and optionally storing them in an environment.

Usage

```
tisFromCsv(csvFile, dateCol = "date8", dateFormat = "%Y%m%d", defaultTif = "busines
           save = F, envir = parent.frame(),
           naNumber = NULL, tolerance = sqrt(.Machine$double.eps), ...)
```

Arguments

`csvFile`	A file name, connection, or URL acceptable to `read.csv`. Also see the the rest of this help entry for required attributes of this file.
`dateCol`	name of the column holding dates. This column must be present in the file.
`dateFormat`	format of the dates in `dateCol`. See `strftime` for date formats.
`defaultTif`	If the frequency can't be inferred from the dates in the `ymdCol` column, this `tif` frequency will be used. This should be a rare occurrence.
`save`	If true, save the individual series in the enviroment given by the `envir` argument. Default is `FALSE`.
`envir`	if `save == TRUE`, the individual series (one per column) are saved in this enviroment. Default is the frame of the caller.
`naNumber`	if non-`NULL`, numbers within `tolerance` of this number are considered to be `NA` values. NA strings can be specified by including an `na.strings` argument as one of the ... arguments that are passed along to `read.csv`.
`tolerance`	Used to determine whether or not numbers in the file are close enough to `naNumber` to be regarded as equal to it. The default is about 1.48e-08.
`...`	Additional arguments passed along to the underlying `read.csv` function.

Details

File Requirements: The csv file must have column names across the top, and everything but the first row should be numeric. There must be as many column names (enclosed in quotes) as there are columns, and the column named by `dateCol` must have dates in the format indicated by `dateFormat`. **The `dateCol` column must be present.**

Missing (NA) values: Missing and NA values are the same thing. The underlying `read.csv` has "," as its default separator and "NA" as its default na.string, so the rows

```
20051231,,13,,42,NA,
20060131,NA,14,,43,,NA
```

indicate `NA` values for both the Dec 2005 and Jan 2006 observations of the first, third, fifth and sixth series.

The values in the file are read into a single large `tis` series, with a `tif` (Time Index Frequency) inferred from the first six dates in the ymd column. The first date is converted to a `ti` (Time Index) of that frequency and becomes the `start` of the series. Each individual column is then windowed via `naWindow` to strip off leading and trailing `NA` values, and the resulting series are put into a list with names given by lower-casing the column names from the csv file. If `save` is `TRUE`, the series are also stored in `envir` using those same names.

Value

A list of `tis` time series, one per column of the csv file. The list is returned invisibly if `save` is `TRUE`.

Author(s)

Jeff Hallman

See Also

`ti`, `tis`, `read.csv`, `read.table`

`tis` *Time Indexed Series*

Description

The function `tis` is used to create time-indexed series objects.

`as.tis` and `is.tis` coerce an object to a time-indexed series and test whether an object is a time-indexed series.

Usage

```
tis(data, start = 1, tif = NULL, frequency = NULL, end = NULL)
as.tis(x, ...)
## S3 method for class 'ts':
as.tis(x, ...)
## S3 method for class 'tis':
as.tis(x, ...)
## Default S3 method:
as.tis(x, ...)
is.tis(x)
```

Arguments

data	a numeric vector or matrix of the observed time-series values.
start	the time of the first observation. This can be a ti object, or anything that ti(start, tif = tif, freq = frequency), can turn into a ti object.
...	other args to be passed to the method called by the generic function. as.tis.default passes x and ... to the constructor function tis.
tif	a ti Frequency, given as either a numerical code or a string. tif() with no arguments returns a list of the allowable numerical codes and names.
frequency	As an alternative to supplying a tif, some tifs can alternatively be specified by their frequency, such as 1 (annual), 2 (semiannual), 4 (quarterly), 6 (bimonthly), 12 (monthly), 24 (semimonthly), 26 (biweekly), 36 (tenday), 52 (weekly), 262 (business) and 365 (daily). Many frequencies have multiple tifs associated with them. For example, all of the tifs (wsunday, wmonday, ..., wsaturday) have frequency 52. In this case, specifying freq gets you the default weekly tif wmonday.
end	the time of the last observation, specified in the same way as start.
x	object to be tested (is.tis) or converted into a tis object. As described in the details below, as.tis can deal with several different kinds of x.

Details

The function tis is used to create tis objects, which are vectors or matrices with class of "tis" and a start attribute that is a ti (time index) object. Time-indexed series are a form of time series that is more flexible than the standard ts time series. While observations for a ts object are supposed to have been sampled at equispaced points in time, the observation times for a tis object are the times given by successive increments of the more flexible time index contained in the series start attribute. There is a close correspondence between Fame time series and tis objects, in that all of the Fame frequencies have corresponding tif codes.

tis objects operate much like vanilla R ts objects. Most of the methods implemented for ts objects have tis variants as well. Evaluate methods(class = "tis") to see a list of them.

One way or another, tis needs to figure out how to create a start attribute. If start is supplied, the function ti is called with it, tif and frequency as arguments. The same process is repeated for end if it was supplied. If only one of start and end was supplied, the other is inferred from

it and the number of observations in data. If both start and end are supplied, the function rep is used to make data the length implied by end - start + 1.

as.tis is a generic function with specialized methods for other kinds of time series. The fallback default method calls tis(x, ...).

Value

tis and as.tis return time-indexed series. is.tis returns TRUE or FALSE.

Note

The tis class is a rewrite of the FRB Splus td class, which itself was based on Jim Berry's TD series idea in Speakeasy.

Author(s)

Jeff Hallman

See Also

Compare with ts. See ti for details on time indexes. cbind.tis combines several time indexed series into a multivariate tis, while mergeSeries merges series, and convert and aggregate convert series from one frequency to another. start.tis and codeend.tis return ti objects, while ti.tis returns a vector ti. There is a print method print.tis and several plotting methods, including lines.tis and points.tis. The window.tis method is also sufficiently different from the ts one to deserve its own documentation.

Examples

```
tis(1:48, start = c(2000, 1), freq = 12)
tis(1:48, start = ti(20000101, tif = "monthly"))    ## same result
tis(0, start = c(2000,1), end = c(2000,52), tif = "weekly")
```

---

today *Time Index for the Current Date*

Description

Usage

```
today()
```

Value

Returns a daily ti for the current date

Author(s)

Jeff Hallman

See Also

ti, Sys.Date

---

t.tis *Matrix Transpose*

Description

Returns the transpose of `as.matrix(x)`

Usage

```
## S3 method for class 'tis':
t(x)
```

Arguments

x	a `tis` object. If `x` is univariate, it will be treated as if it were a single-column matrix, so it's transpose will be a single-row matrix.

Value

A matrix, see `t`. Note that this is **not** a time series.

See Also

t, tis

Examples

```
a <- tis(matrix(1:30, 5,6), start = latestMonth())
a
t(a)  ##i.e.,  a[i, j] == t(a)[j, i] for all i,j, and t(a) is NOT a time series
```

window.tis *Time windows for Time Indexed Series*

Description

window.tis extracts the subset of the object x observed between the times start and end.

Usage

```
window.tis(x, start = NULL, end = NULL, extend = F, ...)
```

Arguments

x	a tis object
start	the start time of the period of interest.
end	the end time of the period of interest.
extend	logical. If true, the start and end values are allowed to extend the series. If false, attempts to extend the series give a warning and are ignored.
...	other arguments to this function are ignored.

Details

The start and end times can be ti objects, or anything that ti(z, tif = tif, freq = frequency), can turn into a ti object.

Value

A tis object that starts and ends at the given times.

Note

The replacement method window<-.tis has not been implemented. Use the subscript operator with a ti argument to replace values of a tis object.

Author(s)

Jeff Hallman

Examples

```
z <- tis(1:24, start = c(2001,1), freq = 12)
z2 <- window(z, start = 19991231, extend = TRUE) ## z2 extends back with NA's
window(z, end = end(z) - 3)
```

ymd *Extract parts of various Date-Time Objects*

Description

Extract the year, month or day, or all three (in yyyymmdd form), or the quarter, from a jul, ti, or from any object that `jul()` can handle.

Usage

```
ymd(x, ...)
## S3 method for class 'jul':
ymd(x, ...)
## S3 method for class 'ssDate':
ymd(x, ...)
## S3 method for class 'ti':
ymd(x, offset = 1, ...)
## Default S3 method:
ymd(x, ...)
year(x, ...)
quarter(x, ...)
month(x, ...)
day(x, ...)
```

Arguments

`x`	a `ti` or `jul`, or something that `jul()` can create a `jul` object from.
`...`	other args to be passed to the method called by the generic function. `year`, `quarter`, `month`, `day` and `ymd.default` may pass these args to `as.Date`.
`offset`	for `ti x`, a number in the range [0,1] telling where in the period represented by `x` to find the day. 0 returns the first day of the period, while the default value 1 returns the last day of the period. For example, if `x` has `tif` = "wmonday" so that `x` represents a week ending on Monday, than any `offset` in the range [0, 1/7] will return the Tuesday of that week, while `offset` in the range (1/7, 2/7] will return the Wednesday of that week, `offset` in the range (6/7, 1] will return the Monday that ends the week, and so on.

Details

`year`, `quarter`, `month` and `day` call `ymd`, and thus understand the same arguments as it does. The default implementation `ymd.default` passes it's arguments to a call to the function `jul`, so all of these functions work the same way that function does.

Value

ymd and it's variants return numeric objects in yyyymmdd form.

year, quarter, month and day return numeric objects.

ymd() with no arguments returns today's yyyymmdd.

Author(s)

Jeff Hallman

See Also

jul, ti, as.Date

Examples

```
ymd()                              ## today's date and time
weekFromNow <- ymd(today() + 7)  ## today() returns a daily ti
year(jul(today()))
month(Sys.time())
## create a monthly tis (Time Indexed Series)
aTis <- tis(0, start = c(2000, 1), end = c(2004, 12), freq = 12)
ymd(ti(aTis))                      ## the yyyymmdd dates of the observations
```

---

alarmc *Interface to alarm() system call*

Description

This is a simple wrapper around the POSIX alarm() system call. Unfortunately, it is not available on Windows.

Usage

```
alarmc(seconds = 0)
```

Arguments

seconds	integer number of seconds to wait before sending SIGALRM to the R process, or 0 (zero) to cancel any existing timer.

Details

alarmc(seconds) sets a timer running for seconds seconds and returns immediately. The default value 0 unsets the timer, if there is one.

When the time is up, the SIGALRM signal is sent to the R process. Since R does not define a handler for this signal, the default handler is invoked, which terminates the R process.

Note

Microsoft, in it's infinite wisdom, does not provide the `alarm()` system call in its C libraries. Instead, they expect you to use the `SetTimer` function to do this kind of stuff, but SetTimer is not available to a console-mode application. I do wonder how they get away with claiming POSIX compatibility.

Author(s)

Jeff Hallman

References

See the man page for alarm

See Also

`Sys.sleep`

Examples

```
## Not run:
## wait up to 10 minutes for a socket connection or die
alarmc(600)
conn <- socketConnection(port = 40001, server = T, blocking = T)
alarmc(0)  ## turn off the timer
incomingText <- readLines(conn)
close(conn)
## End(Not run)
```

`lowLevelFame` *Low Level Fame Interface Functions*

Description

These are most of the lower level functions used in the FAME interface. Most users will never need any of these functions, as the higher level function `getfame` and `putfame` do almost everything they want to do. The functions documented here were written in the course of implementing `getfame` and `putfame`, and some of them may prove useful on their own.

`fameRunning` answers `TRUE` if there is a process called "FAME SERVER" already running under the user's id and with the current R process as its parent process.

`fameStart` initializes the FAME HLI and opens a work database. Since the work database is always the first one opened, its key is always 0.

`fameStop` kills the HLI session and the FAME SERVER process started by `fameStart`. In any given R session, you cannot restart the HLI once it has died for any reason. (This is a FAME limitation, not an R one.) Death of the R process also kills the child FAME SERVER process. So it rarely makes sense to call `fameStop` explicitly, as it makes any subsequent FAME interaction in the current R session impossible.

fameCommand sends its string argument to the child FAME SERVER process to be executed. If silent is TRUE, it invisibly returns a status code that can be sent to fameStatusMessage to get an error message. If silent is FALSE, the status message is echoed to standard output.

fameStatusMessage looks up and returns the error message associated with its argument.

fameDbOpen opens the named database in the given access mode. It returns an integer dbKey, which is a required argument for some of the other functions documented here.

fameDbClose closes the database associated with the given dbKey.

fameDeleteObject deletes the named object from the database associated with the give dbKey.

fameWriteSeries writes the tis (Time Indexed Series) object ser as fname in the database associated with dbKey. If an object by that name already exists in the database and update is TRUE, the frequency and type of ser are checked for consistency with the existing object, and if checkBasisAndObserved is TRUE (not the default), those items are also checked. Any inconsistencies cause the update to fail. If all checks are OK, then the range covered by ser is written to the database. If update if FALSE, any existing series called fname in the database will be replaced by ser. This function should probably not be called directly, as putfame provides a nicer interface.

fameWhat returns a list of low level information about an object in a database, including components named status, dbKey, name, class, type, freq, basis, observ, fyear, fprd, lyear, lprd, obs, and range. If getDoc is TRUE, it will also include description and documentation components. See the FAME documentation for the CHLI functions cfmwhat and cfmsrng for details.

getFamePath finds a path name to a database matching its argument. The default implementation returns the argument if it is a valid path to an existing file, or NULL if it is not. You should create a local implementation of this function if you have some other way to find a valid path to a database. For example, at the Federal Reserve Board, some of our FAME databases are "registered", i.e., their paths can be looked up via a locally written shell script. Our local getFamePath calls that script and returns the full path name to the database, or NULL if something isn't right.

fameLocalInit is a function that is called by fameStart immediately after opening the work database. The default implementation does nothing. You can create a local function with the same name that does additional initialization.

Usage

```
fameRunning()
fameStart()
fameStop()
fameCommand(string, silent = F)
fameStatusMessage(code)
fameDbOpen(dbName, accessMode = "read")
fameDbClose(dbKey)
fameDeleteObject(dbKey, fname)
fameWriteSeries(dbKey, fname, ser, update = F, checkBasisAndObserved = F)
fameWhat(dbKey, fname, getDoc = F)
getFamePath(dbString)
fameLocalInit()
```

Arguments

string	a FAME command to be executed
silent	run the command quietly if TRUE
code	an integer status code from FAME
dbName	name of or path to the database to open
accessMode	a string specifying the access model to open the database in: one of "read", "create", "overwrite", "update", or "shared".
dbKey	integer returned by dbOpen
fname	name of an object in a FAME database
ser	a tis time series
update	if TRUE update any existing series by the same name in place. If FALSE, replace existing series.
checkBasisAndObserved	see description above for fameWriteSeries
getDoc	if TRUE, also return the description and documentation attributes.
dbString	an existing path name or the name of a registered database.

Value

fameRunning return a Boolean.

fameStart and fameStop return nothing.

fameCommand invisibly returns a status code.

fameStatusMessage returns a message string.

fameDbOpen returns an integer dbKey.

fameDbClose returns a status code.

fameDeleteObject returns a status code.

fameWriteSeries returns a status code.

fameWhat returns a list.

getFamePath returns the path to a database or NULL if no match was found.

Author(s)

Jeff Hallman

See Also

getfame, putfame

`ssh` *Execute a command on another machine*

Description

This function composes an ssh (Secure SHell) command to run something on another machine and invokes it via `system()`.

Usage

```
ssh(command, host = getOption("remoteHost"), user. = user(), wait = F, ...)
```

Arguments

`command`	command to be executed on the remote machine
`host`	hostname of the remote machine
`user.`	username on the remote machine
`wait`	if `TRUE`, return only after `command` has finished running on the remote system. If `FALSE` (the default), return immediately after sending `command` to the remote system.
`...`	additional arguments passed to `system`

Details

Uses the ssh program on Unix, and `plink -ssh` on Windows.

Value

The return value is whatever the `system()` function returns. If `...` includes `intern = T`, this will be whatever the ssh or plink returned.

Note

This is a very simple-minded implementation. I did just enough work on it to get the `startRemoteServer` function working and quit while I was ahead. No error checking is done. The Linux version assumes that the `ssh` command is available on your path, while the Windows implementation makes the same assumption about the `plink` program.

Author(s)

Jeff Hallman

Examples

```
## Not run:
ssh("uname -a", host = "localhost")
## End(Not run)
```

Index