Package 'splusTimeDate'

July 7, 2022

Version 2.5.4

Title Times and Dates from S-PLUS

Depends R (>= 2.12.0), methods, stats

Description A collection of classes and methods for working with times and dates. The code was originally available in S-PLUS.

License BSD_3_clause + file LICENSE

URL https://github.com/spkaluzny/splusTimeDate

BugReports https://github.com/spkaluzny/splusTimeDate/issues

NeedsCompilation yes

Author Stephen Kaluzny [aut, cre], TIBCO Software Inc. [aut, cph]

Maintainer Stephen Kaluzny <spkaluzny@gmail.com>

Repository CRAN

Date/Publication 2022-07-07 16:20:02 UTC

R topics documented:

`days` *Return Various Periods from a Time or Date Object*

Description

Creates an ordered factor from a vector of dates according to various calendar periods.

Usage

```
days(x)
weekdays(x, abbreviate = TRUE)
months(x, abbreviate = TRUE)
quarters(x, abbreviate = TRUE)
years(x)
```

Arguments

x	a dates object.
abbreviate	a logical flag. If TRUE, abbreviations should be returned for days of the week, month names, or quarters. Default is TRUE.

Details

- The levels of days are the days of the month, 1 through 31.
- The levels of weekdays are Sunday through Saturday.
- The levels of months are January through December.
- The levels of quarters are I < II < III < IV if abb=F and 1Q < 2Q < 3Q < 4Q if abbreviate=T.
- The levels of years are exactly the years of the dates in x. Years in the range of x that are not in x itself are not interpolated.

These functions are generic.

The default method works on dates objects and other objects that can be coerced to dates objects.

There are also methods for timeDate, which take weekday and month names and abbreviations from timeDateOptions(c("time.day.name","time.day.abb", "time.month.name", "time.month.abb")).

Value

returns an ordered factor corresponding to days, weekdays, months, quarters, or years of x for the respective function.

See Also

as.Date, julian, timeDate, hours, yeardays, hms

Examples

```
## Creates a sample dates object of julian dates
x <- c(10, 11, 12, 42, 44, 45, 101, 102, 212, 213, 214, 300)
dts <- as.Date(x, origin="1960-1-1")
## Alternatively, create a sample timeDate object
dts <- as(x, "timeDate")
## identifies a weekday or month with each julian day
weekdays(dts)
months(dts)
## Produces barplot of # of days in x appearing on particular
## weekday or month
# plot(weekdays(dts))
# plot(months(dts))
## Produces boxplots of julian date by weekday or month
# plot(weekdays(dts), x)
# plot(months(dts), x)
```

format.timeDate *Sample Formats*

Description

The format.timeDate and format.timeSpan data sets are lists with sample formats for the timeDate and timeSpan classes, respectively.

- The names on the lists are the sample strings for each format.
- The corresponding elements are lists whose component is the output format that generates the sample string, and whose input component is the input format that reads in the sample string.

See class.timeDate for information on input and output string formats.

See Also

timeDate class, timeSpan class.

groupVec *groupVec Constructor*

Description

Constructor function for groupVec class objects.

Usage

```
groupVec(names, classes, columns)
```

Arguments

names	the column names for the object. The function can be called with no arguments, but if it is called with any arguments, names must be supplied.
classes	the column classes for the object. If not supplied, it is initialized as ANY, replicated to the length of names.
columns	the list for column data. If not supplied, it is initialized to new(class) for each class in classes.

Details

This function creates a new groupVec object, putting the names (if supplied) into the names slot, the classes (if supplied) into the classes slot, and the columns (if supplied) into the columns slot.

Value

returns a new groupVec object constructed from the arguments, or an empty one if no arguments are supplied.

See Also

groupVec

Examples

```
groupVec()
groupVec(c("my.nums", "my.chars"), c("numeric", "character"))
groupVec(c("my.nums", "my.chars"), c("numeric", "character"),
  list(c(1, 2, 3), c("a", "b", "c")))
```

---

groupVec-class *Group Vector Class*

Description

groupVec is a class that groups a list of vectors so that they can act like a single atomic vector.

Details

groupVec is a class for objects that act as vectors but are actually made up of a collection of vectors of the same length, and which are accessed together and would usually be expected to be printed as if they were simple atomic vectors.

The groupVec class manages the basic vector operations, such as subscripting and length. Extending classes must manage the meaning of the collection of vectors and more specific operations such as character coercion. Extending classes can have other information (although probably not the same length) in other slots. Subscripting on a groupVec-extended class carries this information along unchanged.

A groupVec object also has the following characteristics:

- A valid groupVec object holds a list of vector "columns" with names and classes, where the vectors in the list correspond one-to-one with the names and classes.
- Each vector satisfies an is relationship with its class.
- All the vectors are the same length.

The object is intended to be used as a base class for defining other classes with fixed column names and classes but could also be used as a more dynamic list.

The difference between a groupVec and a data.frame, matrix, or other rectangular structure is that a groupVec is meant to hold objects whose columns really should not be considered as separate observations. For example, the timeDate class is a groupVec whose columns are the date and time of day in GMT, with additional slots for output format and time zone. Because the date can affect the time of day in the local time zone (if there is daylight savings time), and the time of day can affect the date in the local time zone (if the time zone conversion crosses a date boundary), and because each time and date in the timeDate object is displayed as a single character string containing both time and date, the time and date columns are not meant to be accessed separately, and in fact the user does not even have to know that they are stored separately.

The objects in groupVec columns do not have to be atomic vectors. They can be any class that has a well-defined length. This design allows one groupVec object to hold other groupVec objects, lists, data.frames, any type of R vectors, and user-defined types in its columns.

If type checking is not desired for a column, the column class can be set to "ANY", because any R object has an is relationship with the special "ANY" class.

Objects from the Class

Create objects using calls of the form new("groupVector", ...) or groupVector.

Slots

columns (list) the list of vector columns.

names (character) the column names.

classes (character) the column classes.

GroupVec functions

- groupVec: The groupVec constructor function.
- groupVecValid: a groupVec class validity function.
- groupVecExtValid: a validity function that user-defined extending classes with set column names and column classes can use for checking validity.

Although the slots of a groupVec can be accessed directly, it is not recommended. Instead, there are several access functions that you can use on the left or right side of assignments, and that preserve the validity of groupVec objects. These functions are:

- groupVecColumn, for accessing a single column.
- groupVecData, for accessing the entire data list.
- groupVecNames, for accessing the column names.
- groupVecClass, for accessing the column classes.

See individual function documentation for more information.

Methods

Several methods have been defined for groupVec objects for standard vector operations. All operate on groupVec objects as if they were atomic vectors:

- subscripting with [] and [[]] on the left or right side of an assignment.
- length on the left or right side of an assignment.
- c, and is.na

See Also

groupVecValid, groupVecExtValid, groupVec function, groupVecColumn, groupVecData, groupVecNames, groupVecClasses.

groupVecClasses groupVec *Class Data Access*

Description

Accesses or replacees the column classes vector of a groupVec object.

Usage

```
groupVecClasses(object)
```

Arguments

object	the object to access.

Details

This function returns the classes slot of object.

It can also be used on the left side of an assignment to replace the classes slot with a new value. In that case, the data in the object columns are coerced to the new classes. Also, replacement of the classes with a vector of a different length causes the names and columns slots of object to be extended or truncated to the new length.

Value

returns a vector of classes of object data.

See Also

groupVecColumn, groupVecNames, groupVecData, groupVec class.

Examples

```
obj <- new("groupVec")
groupVecClasses(obj) <- c("numeric", "character")
groupVecClasses(obj)
```

groupVecColumn groupVec *Class - Data Access*

Description

Accesses the columns of a groupVec object.

Usage

```
groupVecColumn(object, col.name)
```

Arguments

object	the object to access.
col.name	the name of columns to access.

Details

The function finds the given column by comparing col.name to the names slot of object, and then extracts the corresponding vectors from the list in the columns slot of object.

You can use this function on the left side of an assignment to replace the given columns with new values.

- If you replace one column, supply the value as a vector.
- If you replace multiple columns, supply the values in a list.

The new data is coerced to the columns class using as before the replacement, and the column lengths are checked.

Value

returns the data in the given columns.

- If only one column is supplied, the result is the column
- If more than one name is supplied, the result is a list of columns.

See Also

groupVecData, groupVecNames, groupVecClasses, groupVec class.

Examples

```
obj <- new("groupVec")
groupVecNames(obj) <- "colname1"
groupVecColumn(obj, "colname1") <- c(1, 2, 3)
groupVecColumn(obj, "colname1")
#[1] 1 2 3
```

groupVecData groupVec *Class Data Access*

Description

Accesses the data list of a groupVec object.

Usage

```
groupVecData(object)
```

Arguments

object	the object to access.

Details

The function returns the columns slot of object.

You can use it on the left side of an assignment, in which case the columns slot is replaced, with some validity checking. Also, if the new value has a different length than the old one, the column names and classes are extended or truncated appropriately, with the column classes for new columns derived from the class of the new data in the columns.

Value

returns the data list of object.

See Also

groupVecColumn, groupVecNames, groupVecClasses, groupVec class.

Examples

```
obj <- new("groupVec")
groupVecData(obj) <- list(c(1,2,3), c("a", "b", "c"))
groupVecData(obj)
```

groupVecExtValid groupVec *Extended Class Validation*

Description

Checks the validity for classes that extend the groupVec class.

Usage

```
groupVecExtValid(object, names, classes, checkrest = FALSE)
```

Arguments

object	the object to be validated.
names	a character vector containing correct column names.
classes	a character vector containing correct column classes.
checkrest	a logical value. If TRUE (the default), checks that the non-groupVec slots have length <= 1.

Details

This function checks to see whether an object is a valid groupVec extending object. These are the steps in this process:

1. The groupVecValid function is called to verify that object is a valid groupVec object.
2. The column names in the names slot of object are checked against the names argument, and the column classes in the classes slot of object are checked against the classes argument.
3. If checkrest is true, the groupVecNonVec function is called to check whether the non-groupVec slots of object have length <= 1.

Value

returns TRUE if object is valid; otherwise returns a descriptive string.

See Also

groupVecValid, groupVecNonVec, groupVec class

Examples

```
setClass("myclass", representation(a = "numeric"), contains="groupVec",
   prototype=prototype(names="nums", classes="numeric",
      columns=list(numeric(0)), a=numeric(0)))
setValidity("myclass",
   function(object) groupVecExtValid(object, "nums", "numeric", FALSE))
obj <- new("myclass")
obj@a <- 1:5
validObject(obj)
groupVecExtValid(obj, "nums", "numeric", TRUE)
```

groupVecNames groupVec *Class Data Access*

Description

Accesses the column names of a groupVec object.

Usage

```
groupVecNames(object)
```

Arguments

object	the object to access.

Details

The function returns the names slot of the object.

You can use it on the left side of an assignment to replace the names vector with a new value. In that case, the replacement value is coerced to class character using as. Replacing the names with a vector of a different length causes the classes and columns slots of object to be extended or truncated to the new length. (The default class is numeric for the extension.)

Value

returns a vector of the column names of the object data.

See Also

groupVecColumn, groupVecClasses, groupVecData, groupVec class

Examples

```
obj <- new("groupVec")
groupVecNames(obj) <- c("colname1", "colname2")
groupVecNames(obj)
```

groupVecNonVec groupVec *Extended Class Validation*

Description

Checks whether all slots of an object are either not vectors or have `length` <= 1 for use in validation checks.

Usage

```
groupVecNonVec(object, exceptSlots)
```

Arguments

`object`	the object whose slots are to be checked.
`exceptSlots`	if present, do not check these slots.

Details

The function loops through all the slots of the object, and for each one checks to see that the data in it is either not a `vector` class, or if it is a `vector` class, that it has `length` <= 1.

Value

returns `TRUE` if the slots in the object are atomic or are not `vector` objects (excluding the slots in `exceptSlots`); otherwise, returns a descriptive string.

See Also

`groupVecExtValid`.

Examples

```
setClass("myclass", representation("groupVec", a = "numeric"))
obj <- new("myclass")
groupVecNonVec(obj)
obj@a <- 1:5
groupVecNonVec(obj)
groupVecNonVec(obj, "a")
```

groupVecValid groupVec *Object Validation*

Description

Checks the validity for groupVec objects.

Usage

```
groupVecValid(object)
```

Arguments

object	the object to be validated.

Details

This function validates a groupVec object by checking whether:

1. The object is actually a groupVec (or extending class).
2. The length of the names slot matches the length of the classes slot, and that both match the list length of the columns slot.
3. The classes of the vectors comprising the columns slots list have an is relationship with the corresponding class names from the classes slot.
4. All the vectors in the columns slot list are the same length.

Value

returns a logical value. If TRUE, the object is valid; otherwise, it returns a descriptive string.

See Also

groupVecExtValid, groupVec class

Examples

```
obj <- new("groupVec")
groupVecValid(obj)
```

`holiday.AllSaints` *Holiday Generating Functions*

Description

Generate specific holidays.

Usage

```
holiday.AllSaints(years)
holiday.Anzac(years)
holiday.Australia(years)
holiday.Bastille(years)
holiday.Canada(years)
holiday.Christmas(years)
holiday.Columbus(years)
holiday.Easter(years)
holiday.GoodFriday(years)
holiday.Independence(years)
holiday.Labor(years)
holiday.MLK(years)
holiday.May(years)
holiday.Memorial(years)
holiday.NewYears(years)
holiday.NYSE(years)
holiday.Presidents(years)
holiday.Remembrance(years)
holiday.StPatricks(years)
holiday.Thanksgiving(years)
holiday.Thanksgiving.Canada(years)
holiday.USFederal(years)
holiday.VE(years)
holiday.Veterans(years)
holiday.Victoria(years)
```

Arguments

`years`	the years for which to generate holidays (for example, 1998:2005).

Details

holiday.AllSaints All Saints Day: November 1st.

holiday.Anzac Anzac Day: April 25th.

holiday.Australia Australia Day: January 26th.

holiday.Bastille Bastille Day: July 14th.

holiday.Canada Canada Day: July 1st.

holiday.Christmas Christmas Day: December 25th.

holiday.Columbus Columbus Day: the 2nd Monday in October.

holiday.Easter Easter according to the Roman Catholic tradition.

holiday.GoodFriday Good Friday according to the Roman Catholic tradition.

holiday.Independence U.S. Independence Day: July 4th.

holiday.Labor U.S. Labor Day: the 1st Monday in September.

holiday.MLK U.S. Martin Luther King Jr. Day: the 3rd Monday in January.

holiday.May May Day, also known as Labour Day in some countries: May 1st.

holiday.Memorial U.S. Memorial Day: the last Monday in May.

holiday.NewYears New Years Day: January 1st.

holiday.NYSE New York Stock Exchange holidays, 1885-present, according to the historical and current (as of 1998) schedule, not including special-event closure days or partial-day closures.

holiday.Presidents U.S. Presidents Day: the 3rd Monday in February.

holiday.Remembrance same as holiday.Veterans.

holiday.StPatricks St. Patrick's Day: March 17th.

holiday.Thanksgiving U.S. Thanksgiving Day: the 4th Thursday in November.

holiday.Thanksgiving.Canada Canadian Thanksgiving Day: the 2nd Monday in October.

holiday.USFederal all the U.S. Federal holidays, which are New Years, MLK, Presidents, Memorial, Independence, Labor, Columbus, Veterans, Thanksgiving, and Christmas, all moved to the nearest weekday if they fall on a weekend.

holiday.VE Victory in Europe day: May 8th.

holiday.Veterans November 11th, known as Veterans Day in the U.S. and Remembrance Day in some other countries.

holiday.Victoria Canadian Victoria Day: the Monday on or preceeding the 24th of May.

Value

returns a time/date object with the given holidays. They are not guaranteed to be in any particular order.

In 1971, many U.S. holidays were changed to fall on Mondays, or the holiday(s) began more recently. Because the `holiday` functions always return the modern definition for the holidays, the return values might be inaccurate for dates specified before 1971 or before the beginning date of the holidays. (The exception to that possible inaccuracy is the New York Stock Exchange (NYSE) holidays, which contains their stated holiday schedules as far back as 1885.)

References

The following web sites were used to obtain information on the dates of holidays: http://www.adfa.oz.au/~awm/anzacday/trad
http://www.effect.net.au/cuddlyk/myworld/history/ausday/ausday.html, http://fas.sfu.ca/canheritage/homepage/canid_hp/the
http://www.usis.usemb.se/Holidays/celebrate/intro.htm, http://www.gold.net/~cdwf/misc/easter.html,
http://pages.citenet.net/users/ctmx1108/webcalend/web-cal-top.html http://www.smiley.cy.net/bdecie/Canada.html,
http://shoga.wwa.com/~android7/holidays.htm, http://www.nyse.com

See Also

holiday.fixed, holiday.nearest.weekday, holidays.

Examples

```
holiday.Christmas(1994:2005)
```

---

holiday.fixed *Holiday Generating Functions*

Description

Generates holidays occuring on fixed dates (holiday.fixed) or on given weekdays of given months (holiday.weekday.number), for example, the 1st Monday in September.

Usage

```
holiday.fixed(years, month, day)
holiday.weekday.number(years, month, weekday, index)
```

Arguments

years	the desired year(s) for which to generate holidays (for example, 1997:2000).
month	the month of the holiday (1-12).
day	the day of the month of the holiday (1-31).
weekday	the weekday of the holiday (0-6, 0 is Sunday).
index	the occurrence of weekday in the month (1-5, or -1 for the last) for the holiday.

Details

- The holiday.fixed function generates holidays, like Christmas, that occur on a specified date every year.
- The holiday.weekday.number function generates holidays that occur on an indexed weekday of a given month every year, such as Labor day in the U.S., which is the first Monday of September. Years in which the given dates do not exist (for example, the 5th Friday in various months) are excluded from the output (as opposed to generating NA).

Value

returns a time/date object containing the specified holiday in the specified years.

See Also

holiday.AllSaints, holidays, holiday.nearest.weekday

Examples

```
# Generate Christmas
holiday.fixed(1994:2005, 12, 25)
# Generate Memorial Day (last Monday in May)
holiday.weekday.number(1994:2005, 5, 1, -1)
# Generate Thanksgiving (4th Thursday in November)
holiday.weekday.number(1994:2005, 11, 4, 4)
```

`holiday.nearest.weekday` *Holiday Generating Functions*

Description

Moves calculated holiday dates (or any dates) to the nearest weekday, if they are on weekends.

Usage

```
holiday.nearest.weekday(dates.)
```

Arguments

`dates.`	the dates to move to the nearest weekday. Must be a `timeDate` object.

Value

returns a vector of dates that is the same as the input vector wherever its dates lie on weekdays, and which has Sundays moved to Monday, and Saturdays moved to Friday for weekend dates.

See Also

`holiday.fixed`, `holidays`.

Examples

```
holiday.nearest.weekday(holiday.Christmas(1994:2005))
```

holidays *Holiday Generating Function*

Description

Generates a collection of holidays.

Usage

```
holidays(years, type = "USFederal", move = FALSE)
```

Arguments

years	the years for which to generate holidays.
type	the names of holidays to generate.
move	a logical value. If TRUE, move the holidays to the nearest weekday.

Details

This function calls the holiday.xxx functions, where xxx takes on the value of each of the strings in the type argument. If these functions do not exist, an error occurs.

After calling the holiday.xxx functions, holidays calls holiday.nearest.weekday if move is TRUE. This moves the holidays so they occur on weekdays. move can also be given as a logical vector, in which case each element applies to the corresponding element of type.

Value

returns a time/date object containing the specified holidays, in chronological order. The time of day in the returned value is midnight in the time zone given by timeDateOptions("time.zone").

See Also

holiday.AllSaints, holiday.nearest.weekday, holiday.fixed.

Examples

```
## Generate Christmas, New Years, and Veterans day, moving Christmas
## and New Years to the nearest weekday
holidays(1994:2005, c("Christmas", "NewYears", "Veterans"),
 c(TRUE, TRUE, FALSE))
```

hours *Return Various Periods from a Time Vector*

Description

Extracts as numbers various time-of-day periods from a time vector.

Usage

```
hours(x)
minutes(x)
seconds(x)
yeardays(x)
```

Arguments

x	the timeDate object from which to extract periods.

Value

returns a numeric vector of hours, minutes, seconds, or year-days of x for the respective function.

- Hours are integers between 0 and 23.
- Minutes are integers between 0 and 59.
- Seconds are numbers including the fractions of a second.
- Yeardays are the day number of the year (a number between 1 and 366).

See Also

days, hms.

Examples

```
x <- timeDate(c("1/1/1998 3:05:23.4", "5/10/2005 2:15:11.234 PM"))
hours(x)
minutes(x)
seconds(x)
yeardays(x)
```

is.monthend *The End of Month Day Information*

Description

Indicates whether a "timeDate" object falls on a month end.

Usage

```
is.monthend(x)
```

Arguments

x	a "timeDate" object

Value

returns a logical vector indicating which element of x is an end-of-month day.

See Also

holidays, weekdays.

Examples

```
is.monthend(timeDate(c("1/1/1958", "1/31/1958", "2/10/1958")))
```

mdy *Return Various Periods from a Time Vector*

Description

Returns data frames containing various periods from a time vector as integers.

Usage

```
mdy(x)
hms(x)
wdydy(x)
```

Arguments

x	the time object to extract calendar periods from.

Details

- mdy returns a data frame with three columns:
  - month (1 - 12).
  - day (1 - 31).
  - year (for example, 1998).
- hms returns a data frame with four columns:
  - hour (0 - 23).
  - minute (0 - 59).
  - second (0 - 59).
  - ms (0 - 999).
- wdydy returns a data frame with three columns:
  - weekday (0 - 6, with 0 meaning Sunday and 6 meaning Saturday).
  - yearday (0 - 366).
  - year (for example, 1998).

Value

returns a data frame containing the periods as integers.

See Also

hours, days.

Examples

```
x <- timeDate(c("1/1/1998 3:05:23.4", "5/10/2005 2:15:11.234 PM"))
mdy(x)
hms(x)
wdydy(x)
```

---

numericSequence *Constructor for* numericSequence *Class*

Description

Constructs numericSequence objects.

Usage

```
numericSequence(from, to, by, length.)
```

Arguments

from	start of the sequence.
to	end of the sequence.
by	increment for the sequence.
length.	length of the sequence; a non-negative integer.

Details

At least three of the four arguments must be supplied, unless the function is called with no arguments.

Value

returns a numericSequence object with properties given by the arguments, or the default numericSequence if no arguments are supplied.

See Also

numericSequence class.

Examples

```
numericSequence()
# The following all produce a sequence running from 1 to 10:
numericSequence(1, 10, 1)
numericSequence(1, by = 1, length = 10)
numericSequence(1, 10, length = 10)
numericSequence(to = 10, by = 1, length = 10)
```

---

numericSequence-class *Numeric Sequence Class*

Description

The numericSequence class is a compact representation of a numeric vector in an arithemetic sequence.

Details

The numericSequence class extends the positionsNumeric class.

Valid numericSequence objects must contain a single non-NA number in at least three of the four slots. If all four are present, the length slot is ignored, and a warning message is generated when the sequence is used.

If length is present and not ignored, it must be non-negative. (A zero-length sequence is equivalent to numeric(0).) Otherwise, the sign of the by slot must agree with the sign of (to - from) to have

a valid sequence. In particular, if by is zero, then to and from must be equal. The default sequence (generated by calling numericSequence() or new("numericSequence")) has length 0.

A numericSequence can be coerced to numeric or integer using as, and regularly-spaced numbers can be coerced to numericSequence using as. This fails if the input is not a regular arithmetic sequence within a tolerance given by timeDateOptions("ts.eps").

Most operations on numericSequence objects (for example, mathematical functions, arithmetic, comparison operators, or subscripting) work by first coercing to a numeric vector, and therefore do not return numericSequence objects.

Slots

from (numeric) the start of the sequence.

to (numeric) the end of the sequence.

by (numeric) the increment for the sequence.

length (integer) the length of the sequence.

See Also

numericSequence function.

---

positions-class *Virtual Classes for Time-Related Objects*

Description

The positions, positionsNumeric, and positionsCalendar classes are virtual classes that represent positions for series objects. The timeInterval class represents time intervals.

Details

- The positionsNumeric class is a class union of numericSequence and numeric, allowing representations of numeric positions either by a sequence object or simply a vector.
- The positionsCalendar is a VIRTUAL class and is extended by timeDate and timeSequence.
- The positions class is a class union of positionsNumeric and positionsCalendar.
- The timeInterval class is extended by timeSpan and timeRelative.

See Also

timeDate class and timeSequence class.

shiftPositions *Shift a Positions Object*

Description

Returns a positions object similar to the input but shifted in time.

Usage

```
shiftPositions(x, k=1)
```

Arguments

x	a vector or timeDate object. Missing values (NAs) are allowed.
k	the number of positions the input series is to shift for the new series. • If k is a positive value, the resulting series is shifted forwards in time. • If k is a negative value, the resulting series is shifted backwards in time. If k is a non-integer value, it is rounded to the nearest integer before the shift is applied.

Value

returns a vector or timeDate object with the positions shifted by k steps.

See Also

lag

Examples

```
shiftPositions(1:10, 1)
x <- as(1:10, "timeDate")
shiftPositions(x)
```

timeAlign *Alignment of Times*

Description

Aligns a time vector to a time unit specified as in timeSeq.

Usage

```
timeAlign(x, by="days", k.by=1, direction=1, week.align=NULL,
          holidays=timeDate())
```

Arguments

x	a time/date object.
by	one of the following character strings, giving the units to align to: • "milliseconds" • "seconds" • "minutes" • "hours" • "days", • "weekdays" • "bizdays" • "weeks" • "months" • "quarters" • "years"
k.by	a non-zero integer giving the number of the by units to align to. Ignored for "weekdays", "bizdays", and "weeks".
direction	either 1 or -1, to align to the next or previous time that is an integer number of the k.by * by units.
week.align	if not NULL, and by is "weeks", you can supply a character string (or an integer, 0 to 6 with 0 being Sunday) to specify a weekday to align to. The character string must be sufficient to make a unique case-insensitive match to the strings in timeDateOptions("time.day.name").
holidays	dates of holidays for business day alignment.

Value

returns a time object whose elements are moved up or down (according to direction), so that they lie on integer multiples of k.by * by units of time, starting from the beginning of the next larger time unit (for example, if by="days", then align to multiples of k.by days added to the first of the month. If by="hours", align to multiples of k.by hours since midnight).

Notes

- for "weeks", "weekdays", and "bizdays", k.by is assumed to be 1 and ignored.
- "weeks" without week.align is equivalent to "days".
- k.by should be a divisor of the number of by units in the next larger time unit, or NA values result.

See Also

timeSeq.

Examples

```
x <- timeDate(c("2/11/1992 22:34", "7/8/1995 08:32"),
       format="%a %02m/%02d/%Y %02H:%02M")
# move to beginning of month
timeAlign(x,"months",direction=-1)
# move to beginning of next month
timeAlign(x,"months",direction=1)
# move to next multiple of 3 hours
timeAlign(x,"hours",3)
# move to next Friday
timeAlign(x,"weeks", week.align="Friday")
```

`timeCalendar` *Constructor Function for* `timeDate` *Objects*

Description

Constructs a time object corresponding to calendar dates and/or times of day.

Usage

```
timeCalendar( m=NULL, d=NULL, y=NULL, h=NULL, min=NULL,
   s=NULL, ms=NULL, format, zone )
```

Arguments

`m`	calendar months (1-12). The default is 1.
`d`	calendar days (1-31). The default is 1.
`y`	calendar years (e.g. 1997). The default is 1960.
`h`	hours of the days (0-23). The default is 0.
`min`	minutes of the days (0-59). The default is 0.
`s`	seconds of the days (0-59). The default is 0.
`ms`	milliseconds of the days (0-999). The default is 0.
`format`	the output format string to apply to the returned object. The default is from `timeDateOptions("time.out.format")`.
`zone`	the time zone of the input date and time. Also stored in the result. The default is from `timeDateOptions("time.zone")`.

Details

If none of `m`, `d`, `y`, `h`, `min`, `s`, and `ms` are supplied, this function returns a time vector of length zero. If more than one is supplied, they must all have compatible lengths. Shorter inputs are used cyclically to comprise the maximum length, but they must be even multiples.

The arguments represent the date and time of day in the given time zone. Missing arguments are supplied to give a time of midnight, January 1, 1960.

Leap seconds do not cause `NA` values, but can cause times of day to be off by a second in the days containing them.

Value

returns a timeDate object corresponding to the input.

See Also

timeDate, timeDate class, mdy, format.timeDate.

Examples

```
timeCalendar(m = c(3, 4, 5), d = c(12, 15, 7), y = c(1998, 1997, 2004),
     format = "%b. %d, %Y")
timeCalendar(h = c(9, 14), min = c(15, 23), format = "%I:%02M %p")
```

---

timeCeiling *Rounding Functions for timeDate Objects*

Description

Rounds a time to the nearest day.

Usage

```
timeCeiling(x)
timeFloor(x)
timeTrunc(x)
```

Arguments

x	an object of class positionsCalendar

Value

returns a positionsCalendar object rounded to current or next day.

timeFloor and timeTrunc	round a positionsCalendar object to the beginning of the day.
timeCeiling	rounds a time to the beginning of the next day.

See Also

ceiling, floor, trunc, positionsCalendar-class

Examples

```
x <- timeDate(date(), in.format="%w %m %d %H:%M:%S %Y")
timeCeiling(x)
timeFloor(x)
```

`timeConvert` *Convert from one time zone to another.*

Description

Converts a time/date object from one time zone to another.

Usage

```
timeConvert(x, to.zone, from.zone)
```

Arguments

`x`	the time/date object to convert.
`to.zone`	the time zone to convert to (character).
`from.zone`	the time zone to convert from (character). For `timeDate` objects, this is ignored. Instead, the value is taken from the `time.zone` slot of the `timeDate` object.

Details

The `timeDate` stores times/dates as their equivalent time in GMT, with a time zone (used for printing and other operations) stored in the `time.zone` slot. Therefore, to convert time zones, this function simply puts the new time zone into the `time.zone` slot.

Value

returns the time/date object reflecting the converted time zone.

See Also

`timeZoneConvert`.

Examples

```
timeDateOptions(time.zone="GMT",
  time.in.format="%m/%d/%Y [%H:%M]",
  time.out.format="%m/%d/%Y %02H:%02M (%Z)")
date1 <- timeDate("3/22/2002 12:00", zone="PST")
date1
## 3/22/2002 12:00 (PST)
date2 <- timeConvert(date1, "EST")
date2 # converted to EST
## 3/22/2002 15:00 (EST)
```

`timeDate` *Constructor Function for timeDate Objects*

Description

Constructs a `timeDate` object from a character vector, a vector of julian days, or a vector of milliseconds, or constructs an empty `timeDate` object.

Usage

```
timeDate(charvec, in.format,
         format, zone,
         julian, ms, in.origin=c(month=1, day=1, year=1960))
```

Arguments

`charvec`	a character vector to read the times from.
`in.format`	the input format string for `charvec`. The default value is `timeDateOptions("time.in.format")`. (Click `class.timeDate` in the **SEE ALSO** section for the list of allowable input format strings.
`format`	the output format stored in the result. The default is `timeDateOptions("time.out.format")`.
`zone`	the time zone stored in the result. The default value is `timeDateOptions("time.zone")`.
`julian`	an integer vector of the number of days since `in.origin`. If `ms` is missing, this argument can also be a numeric whose fractional part gives the fraction of the day.
`ms`	an integer vector of milliseconds since midnight.
`in.origin`	the origin for the `julian` argument. This should be a vector with `month`, `day`, and `year` components.

Details

One of `charvec`, `julian`, or `ms` must be supplied, unless the function is called with no arguments.

- If `charvec` is supplied, then `timeDate` reads the times from the `charvec` character strings using the format string from `in.format`. This conversion uses the time zone supplied in `zone`.
- If `charvec` is not supplied, then `timeDate` uses `julian` and/or `ms` to construct the time vector. These are copied directly to the internals of the time object without considering the time zone (that is, they must be given in GMT, or the user must call `timeZoneConvert` afterwards).

Leap seconds do not cause `NA` values, but it can cause times of day to be off by a second on the days that contain them.

For information about possible values for the `in.format` and `format` arguments, see the documentation for the `timeDate` class (`class.timeDate`).

Value

returns a timeDate object derived from the inputs.

See Also

timeCalendar, timeDate, format.timeDate, timeZoneConvert.

Examples

```
timeDate()
timeDate(c("1/1/97", "2/1/97", "mar 1, 1997"))
timeDate(c("1 PM", "2 PM", "3 AM"), in.format = "%H %p",
      format = "%I %p")
timeDate(julian = 36, ms = 876393,
      in.origin = c(month=1,day=1,year=1998))

## Get today's date in yyyymmdd format
timeDate(date(), in.format="%w %m %d %H:%M:%S %Y",
      format="%Y%02m%02d")
```

---

timeDate-class *Time and Date Class*

Description

The timeDate class represents times and dates.

Details

This class holds a vector of times and/or dates. It extends the groupVec class, as well as the positionsCalendar class (see the documentation for the positions class).

The groupVec portion of a time object holds a date portion, stored as a vector of the days since January 1, 1960, and a time portion, stored as a vector of the number of milliseconds since midnight, GMT. The groupVec column names are "julian.day" and "milliseconds", and the column classes are integer.

The string in the time.zome slot must be one of the names from the time zone list (see the timeZoneList documentation). Because times are stored internally always in GMT, the time-zone string is used for printing and for converting to calendar times and dates (month/day/year, hour/minute/second/millisecond). You can change these directly. You can also change the format directly, but we do not recommend changing the groupVec slots directly.

Slots

columns (list) (from groupVec).

names (character) (from groupVec).

classes (character) (from groupVec).

format (character) output format string.

time.zone (character) time zone string.

Time functions

Objects of `class.time` can be created using the `new` function, in which case they are constructed to have length 0 and the default format and zone from `timeDateOptions("time.out.format")` and `timeDateOptions("time.zone")` respectively. Alternatively, they can be created using the `timeDate` and `timeCalendar` functions.

There are `as` relationships set up for `timeDate` objects to coerce them to and from `character`, `numeric`, and `integer`.

For numbers, the integer part is the julian day, and the fractional part is the fraction of the day given by the number of milliseconds divided by the number of milliseconds in a day, in GMT.

Addition of numbers to time objects and subtraction of numerics from time objects works as though the time were converted to a number, the operation were performed, and the number was converted back to a time. Their subtraction results in a `timeSpan` object, and a `timeSpan` object can be added to or subtracted from a time.

Only a few other mathematical functions make sense for time objects: `floor`, `ceiling`, `min`, `max`, `mean`, and `range`. Multiplication, division, and other operations that do not make sense for times and dates (in the absence of an origin) result in numbers, via automatic coercion to class `numeric`.

Note that while conversion to/from numerics is always in GMT, `floor` and `ceiling` take account of the time zone to ouput time objects whose time is midnight in their time zone, and whose date is no later than the input time's date for `floor`, and no earlier for `ceiling`. In addition to these mathematical operations, all of the standard comparison operators have methods for comparing two time objects.

There are also functions for time objects that pick out particular parts. See `days`, `hours`, and `mdy` for more information.

Various options are used by the time class, primarily for printing to and reading from character strings. See `timeDateOptions` for documentation.

Formatting

The input and output format specifications look familiar to C and IMOX programmers and are patterned on the `strptime` function under Solaris.

Input formats

Input format strings are used to convert character strings to time objects. When reading time objects, the default of January 1, 1960, Midnight GMT is supplied, and the input format specifications below can be used to override this default time. They are read in from left to right. If the entire input string is not matched by the format string, or if the resulting time or date is not valid, an `NA` is inserted into the time vector. (To skip characters in a string, use %c or %w.)

NOTE: If you are reading a time zone from your character string, the notation used for the time zone in your character string must be one of the components of the time zone list. See documentation for `timeZoneList` for more information.

***** anything not in this list matches itself explicitly.

%c any single character, which is skipped. This is useful for skipping entries like days of the week, which, if abbreviated, could be skipped by "%3c" (see also %w). To skip the rest of the string, use "%$c".

%d input day, within a month, as integer.

%H input hour as integer.

%m input month as integer or as alpha string. If an alpha string, case does not matter, and any substring of a month in `timeDateOptions("time.month.name")` that distinguishes it from the other months is accepted.

%M input minute as integer.

%n input milliseconds as integer, without considering field width as in %N.

%N input milliseconds as integer. A field width (either given explicitly or inferred from input string) of 1 or 2 causes input of 10ths or 100ths of a second instead, as if the digits were following a period. Field widths greater than 3 are likely to result in illegal input.

%p accepts strings from `timeDateOptions("time.am.pm")`, defining `am` and `pm`, with matching as for months. If `pm` is given, and the hour is before 13, the time is bumped into the afternoon. If `am` is given, and the hour is 12, the time is bumped into the morning. Note that this modifies previously-parsed hours only.

%S input seconds as integer.

%w a whitespace-delimited word, which is skipped (no width or delimiter specification. For that, use %c).

%y input year as integer. If less than 100, `timeDateOptions("time.century")` is used to determine the actual year.

%Y input year as integer, without considering the century.

%Z a time zone string. Accepts a whitespace-delimited word, unless another delimiter or width is specified. The legal time zones are the names of `timeZoneList()`.

%(digits)(char) if there are one or more digits between "%" and the specification character, these are parsed as an integer, and specify the field width to be used. The following (digits) characters are scanned for the specified item.

%:(delim)(char) if there is a colon and any single character between a "%" and the specification character, the field is taken to be all the characters up to but not including the given delimiter character. The delimiter itself is not scanned or skipped by the format.

%$(char) If there is a $ between a % and a specification character, the field goes to the end of the input string.

whitespace whitespace (spaces, tabs, carriage returns, and so on) is ignored in the input format string. In the string being parsed, any amount of white space can appear between elements of the date/time. Thus, the parse format string " %H:%M: %S " parses "5: 6:45".

[...] specify optional specification. Text and specifications within the brackets optionally can be included. This does not support fancy backtracking between multiple optional specs.

%%,%[,%] the %, [, and] characters, which must be matched in the input string.

Output formats

Output formats are used to convert time objects to character strings. They are stored in the format slot of the time object. During output, if a given field width is too short to hold the output, if that output field is a character field, the leftmost characters are printed, but if it is a numeric field, the output string becomes "NA". The following format specifications can be used:

* anything not in this list matches itself explicitly (including whitespace, unlike in input specs).

%a print abbreviated weekday ("Mon", and so on) from timeDateOptions("time.day.abb").

%A print full weekday ("Monday", and so on) from timeDateOptions("time.day.name").

%b print month as abbreviation, from timeDateOptions("time.month.abb").

%B print month as full name, from timeDateOptions("time.month.name").

%C print year within century as integer: 0-99.

%d print day within month as integer: 1-31.

%D print day within year as integer: 1-366.

%H print hour (24-hour clock) as integer, 0-23.

%I print hour (12-hour clock) as integer, 1-12.

%m print month as integer: 1-12.

%M print minutes as integer: 0-59.

%N print milliseconds as integer. It is a good idea to pad with zeros if this is after a decimal point! A width of less than 3 causes printing of 10ths or 100ths of a second instead: 0-999.

%p print "am" or "pm", using strings from timeDateOptions("time.am.pm").

%q print quarter of the year, as integer: 1-4.

%Q print quarter of the year, as Roman numeral: I-IV.

%S print seconds as integer: 0-59 (60 for leap second).

%y print year as two-digit integer if year is in century specified by timeDateOptions("time.century"), otherwise full year.

%Y print full year as integer (see also %C).

%Z print the time zone string from the objects `time.zone` slot.

%z print the time zone string from the objects `time.zone` slot, using the part before the first `"/"` character if it is standard time, and the part after the first `"/"` character if it is daylight savings time (that is, if the time zone is `"PST/PDT"`). If there is no `"/"` character, the entire time zone is used for both.

%% print the % character

%(digits)(char) if there are one or more digits between "%" and the specification character, these are parsed as an integer, and specify the field width to use. The value is printed, right justified using (digits) characters. If (digits) begins with zero, the field is left-padded with zeros if it is a numeric field, otherwise it is left-padded with spaces. If a numeric value is too long for the field width, the field is replaced with asterix "*" characters to indicate overflow; character strings can be abbreviated by specifying short fields.

Note

The calendar follows the conventions of the British Empire, which changed from Julian to Gregorian calendars in September of 1752. Calendar dates prior to 1920 were different in many countries. See the "Calendar FAQ" posted regularly to Usenet news groups soc.history, sci.astro, sci.answers, soc.answers, and news.answers, and to a web site at http://www.pip.dknet.dk/~c-t/calendar.html for more information on the history of calendars around the world. The time objects allow days with leap seconds, but calculated times of day for days containing leap seconds might be off by a second; they are treated as though the leap second occurred at the very end of the day, because there is currenly no provision in the splusTimeDate package for keeping track of leap seconds.

See Also

groupVec class, timeSpan class, timeDateOptions, timeDate function, timeCalendar, format.timeDate.

Examples

```
## The default format for input is initially:
##    "%m[/][.]%d[/][,]%y [%H[:%M[:%S[.%N]]][%p][[(]%3Z[)]]]"
## This allows reading strings such as
##    "Jan 22 1997", "January 22, 1997", "1/22/97", "1/22/97 2PM"
## The default format for output is initially:
##    "%02m/%02d/%Y %02H:%02M:%02S.%03N"
## Another choice would be
##    "%A %B %d, %Y %I:%02M %p"
## These would result in the following output:
##    "01/22/1997 14:34:45.025" and "Thursday January 22, 1997 2:34 PM"
```

---

timeDateOptions *Set or Return* timeDate *Options*

Description

Provides the means to set or view global options for working with timeDate objects and classes.

Usage

```
timeDateOptions(...)
```

Arguments

...	you can give a list or vector of character strings as the only argument, or you can give any number of arguments in the name=value form. Optionally, you can give no arguments. See the **Value** and **Side Effects** sections for explanation.

Details

To see all the timeDate options and their current values, call timeDateOptions with no arguments i.e. timeDateOptions()

To set timeDateOptions temporarily in a function, call timeDateOptions as you normally would from the command line. To ensure that your function finishes cleanly and does not produce any side effects, use on.exit with the return value from your call to timeDateOptions.)

Value

a list, even if the list is of length 1.

- If no arguments are given, timeDateOptions returns a list of current values for all options.
- If a character vector is given as the only argument, timeDateOptions returns a list of current values for the options named in the character vector.

- If an object of mode "list" is given as the only argument, its components become the values for options with the corresponding names. timeDateOptions returns a list of the option values before they were modified. Usually, the list given as an argument is the return value of a previous call to timeDateOptions.
- If arguments are given in name=value form, the values of the specified options are changed and timeDateOptions returns a list of the option values before they were modified.

Common options

ts.eps a small number specifying the time series comparison tolerance. This is used throughout the time series functions for frequency comparisons. Frequencies are considered equal if they differ in absolute value by less than ts.eps.

sequence.tol a number specifying the tolerance for converting numeric vectors to numeric sequences. If a numeric vector is an arithmetic sequence to within sequence.tol, it can be converted to a sequence.

time.in.format a character string specifying the format for reading timeDate objects from character strings using the as and timeDate functions. The default value is "%m[/][.]%d[/][,]%y [%H[:%M[:%S[.%N]]][%p][[(]%3Z[)]]]", which reads a wide variety of date strings. To use the European day/month/year format, set this to "%d[/][.]%m[/][,]%y [%H[:%M[:%S[.%N]]][%p][[(]%3Z[)]]]". The elements of this format string are described in the documentation for the timeDate class (class.timeDate).

time.out.format a character string specifying the format for printing timeDate objects to character strings. The default value is "%02m/%02d/%04Y %02H:%02M:%02S.%03N". To use the European day/month/year format, set this to "%02d/%02m/%04Y %02H:%02M:%02S.%03N". The elements of this format string are described in the documentation for the timeDate class (class.timeDate).

time.out.format.notime a character string specifying the format for printing timeDate objects when the time.zone option is set to GMT and the time of every element of the timeDate object is midnight. See timeDate for more information.

time.month.name a 12-element character vector giving the names of the months.

time.month.abb a 12-element character vector giving the abbreviations for the names of the months.

time.day.name a 7-element character vector giving the names of the days of the week, starting with Sunday.

time.day.abb a 7-element character vector giving the abbreviations for the names of the days of the week, starting with Sunday.

time.century an integer indicating the first year of a 100-year span. This value is used to interpret and print two-digit years. For example, if time.century=1950, the year 50 is interpreted as 1950 and the year 49 is interpreted as 2049. If time.century=1900, the year 0 means 1900 and the year 99 means 1999.

time.am.pm a 2-element character vector giving strings for printing "AM" and "PM" in time objects.

time.zone a character string specifying the default time zone when none is given in a time object.

tspan.in.format a character string specifying the format for reading timeSpan objects from character strings using the as and timeSpan functions.

tspan.out.format a character string specifying the format for printing timeSpan objects to character strings.

Default values

The default values for some of the common options listed above are as follows. Options that have never been set have the value NULL

```
sequence.tol=1e-6
time.am.pm=c("AM", "PM")
time.century=1930
time.in.format="[%m[/][.]%d[/][,]%y] [%H[:%M[:%S[.%N]]][%p][[(]%3Z[)]]]"
time.out.format="%02m/%02d/%Y %02H:%02M:%02S.%03N"
time.day.abb=c("Sun", "Mon", ..., "Sat")
time.day.name=c("Sunday", "Monday", ..., "Saturday")
time.month.abb=c("Jan", "Feb", ..., "Dec")
time.month.name=c("January", "February", ..., "December")
time.zone="GMT"
tspan.in.format=paste("[%yy[ear[s]][,]] [%dd[ay[s]][,]]",
  "[%Hh[our[s]][,]] [%Mm[in[ute][s]][,]] [%Ss[ec[ond][s]][,]]",
  "[%NM[s][S]]")
tspan.out.format="%dd %Hh %Mm %Ss %NMS",
ts.eps=1e-5
```

Side Effects

If timeDateOptions is called with either a list as the single argument or with one or more arguments in name=value form, the options specified are changed or created. The options are stored in a list in a local environment within the splusTimeDate package. Any modifications to the options disappear when the current session ends. The next session will start with the default value of the options.

Examples

```
timeDateOptions(time.zone="PST")
```

---

timeEvent *Constructor Function For* timeEvent *Objects*

Description

Constructs a timeEvent object.

Usage

```
timeEvent(start., end., IDs)
```

Arguments

start.	a time/date object giving start times of events. The function can be called with no arguments, but if any are supplied, start. is required.
end.	a time/date object giving end times of events. If missing, defaults to one day after start., minus 1 millisecond.
IDs	the names or numbers identifying individual events in the object. If missing, defaults to empty strings.

Details

The start., end. and IDs are put into the corresponding columns of a new timeEvent object.

Value

returns a timeEvent object derived from the inputs.

See Also

timeEvent

Examples

```
timeEvent()
timeEvent(holiday.Christmas( 1990:2010 ), ID = 1990:2010)
```

---

timeEvent-class *Event Class*

Description

The timeEvent class represents events that occur at specific calendar times. It is useful for one-time events (for example, the Gulf War), recurring events (for example, holidays or market opening and closing times), and multiple related events (for example, the numbered Olympic games, OPEC meetings, or hurricanes).

Details

The timeEvent class is set up to hold vectors of starting and ending times of events, as well as an identifier for each event, which can be stored in any vector object. These three vectors are stored as columns of a groupVec. The timeEvent class extends the groupVec class.

Create objects of class timeEvent either by using the timeEvent function, or by coercing any positionsCalendar object to timeEvent using as.

Objects from the Class

Create objects using calls of the form new("timeEvent", ...) or timeEvent.

Slots

columns (list) (from groupVec). Always should be a list with three elements.

names (character) (from groupVec). Always c("start", "end", "IDs").

classes (character) (from groupVec). Always c("positionsCalendar", "positionsCalendar", "ANY").

See Also

groupVec class, positionsCalendar class, timeEvent function.

---

timeRelative *Constructor Function for timeRelative Class*

Description

Construct a timeRelative object.

Usage

```
timeRelative(x, holidays., by, k.by=1, align.by=FALSE, week.day=NULL)
```

Arguments

x	a character string vector representing relative times.
holidays.	a time/date or time sequence object giving holiday dates.
by	as an alternate to providing a character string vector, you can provide by, k.by, align.by, and week.day if you need to construct a timeRelative object. by is one of the following character strings, giving the time units for the relative time: "milliseconds" "ms" "seconds" or "sec" "minutes" or "min" "hours" or "hr" "days" or "day" "weekdays" or "wkd" "bizdays" or "biz" "weeks" or "wk" "tdy" (for 10-day periods in a month) "months" or "mth" "quarters" or "qtr" "years" or "yr" To add or subtract specific days of the week, use

"sun"
"mon"
"tue"
"wed"
"thu"
"fri"
"sat"

See timeRelative-class for more information on these arguments.

k.by — a non-zero integer specifying the number of by time units in the relative time.

align.by — a logical value. If TRUE, specifies using alignment (see documentation on the relative time class).

week.day — if not NULL, and by is "weeks", you can supply a character string (or an integer 0 to 6, with 0 being Sunday) to specify a weekday for the relative time. The character string must be sufficient to make a unique case-insensitive match to the strings in timeDateOptions("time.day.name").

Value

returns a timeRelative object with the given strings as data, and holidays, if given. Otherwise, the strings default to empty, and the holidays to no holidays.

See Also

timeRelative

Examples

```
# Create a relative time object that you could add to a time/date object
# to take each element to the third Friday of the month
rtobj <- timeRelative("-a0mth -1fri +3fri")
timeDate(c("1/5/1998", "2/26/1998"), format = "%a %m/%d/%Y") + rtobj
# Create a relative time object for 3 minutes
timeRelative(by="minutes", k.by=3)
```

---

timeRelative-class *Relative Time Class*

Description

The timeRelative class represents relative times.

Details

The `timeRelative` class stores a representation of relative times. Unlike `timeSpan`, which stores absolute time differences, the `timeRelative` class stores relative times in units such as weekdays, months, and business days, whose absolute time value depends on the `timeDate` object with which they are combined.

Both `timeRelative` and `timeSpan` extend the virtual `timeInterval` class.

The `Data` slot in a `timeRelative` object holds a character vector that represents the relative time. Each element of the vector is a character string consisting of whitespace-separated fields in the following form:

```
"[+-][a]#abb"
```

This vector is composed of a required sign (either `"+"` or `"-"`), followed by an optional `"a"` that, if present, means to align the result (see below; it is also possible to specify `0` if aligning), followed by a positive integer and one of the relative time field abbreviations from the following list:

ms add/subtract milliseconds. `"a"` aligns to the nearest # milliseconds within the second, where # must be a divisor of 1000 and less than 1000 (for example, 500 aligns to even seconds or 1/2 seconds). 0 is not allowed.

sec add/subtract seconds. `"a"` aligns to nearest # seconds within the minute, where # must be a divisor of 60 and less than 60 (for example, 15 aligns to 0, 15, 30, or 45 seconds past the minute). 0 goes to the beginning of the current second, independent of sign.

min add/subtract minutes. `"a"` aligns to nearest # minutes within the hour, where # must be a divisor of 60 and less than 60 (for example, 15 aligns to 0, 15, 30, or 45 minutes after the hour). 0 goes to the beginning of the current minute, independent of sign.

hr add/subtract hours. `"a"` aligns to nearest # hours within the day, where # must be a divisor of 24 and less than 24 (for example, 6 aligns to midnight, 6AM, noon, or 6PM). 0 goes to the beginning of the current hour, independent of sign.

day add/subtract days. `"a"` aligns to nearest # days within the month, starting with the first, where # must be a less than the number of days in the month (for example, 2 aligns to the 1st, 3rd, 5th, and so on, with the time midnight). 0 goes to the beginning of the current day, independent of sign.

wkd add/subtract weekdays. `"a"` causes the first added or subtracted weekday possibly to be a fraction of a day to move to the next or previous midnight on a weekday morning, and then whole additional days are added or subtracted to make up # weekdays. 0 goes to the beginning of the day, or the closest weekday before if it is not a weekday, independent of sign.

biz add/subtract business days (weekdays that are not holidays). `"a"` causes the first added or subtracted business day possibly to be a fraction of a day to move the next or previous midnight on a business day morning, and then whole additional days are added or subtracted to make up # business days. 0 goes to the beginning of the day, or the closest business day before if it is not a business day, independent of sign.

sun add/subtract Sundays. `"a"` causes the first added or subtracted Sunday possibly to be a fraction of a day or week to move the next or previous midnight on a Sunday morning, and then whole additional weeks are added or subtracted to make up # Sundays. 0 goes to the beginning of the day, or the closest Sunday before if it is not the right day, independent of sign.

mon add/subtract Mondays. `"a"` causes the first added or subtracted Monday possibly to be a fraction of a day or week to move the next or previous midnight on a Monday morning, and then whole additional weeks are added or subtracted to make up # Mondays. 0 goes to the beginning of the day, or the closest Monday before if it is not the right day, independent of sign.

tue add/subtract Tuesdays. `"a"` causes the first added or subtracted Tuesday possibly to be a fraction of a day or week to move the next or previous midnight on a Tuesday morning, and then whole additional weeks are added or subtracted to make up # Tuesdays. 0 goes to the beginning of the day, or the closest Tuesday before if it is not the right day, independent of sign.

wed add/subtract Wednesdays. `"a"` causes the first added or subtracted Wednesday possibly to be a fraction of a day or week to move the next or previous midnight on a Wednesday morning, and then whole additional weeks are added or subtracted to make up # Wednesdays. 0 goes to the beginning of the day, or the closest Wednesday before if it is not the right day, independent of sign.

thu add/subtract Thursdays. `"a"` causes the first added or subtracted Thursday possibly to be a fraction of a day or week to move the next or previous midnight on a Thursday morning, and then whole additional weeks are added or subtracted to make up # Thursdays. 0 goes to the beginning of the day, or the closest Thursday before if it is not the right day, independent of sign.

fri add/subtract Fridays. `"a"` causes the first added or subtracted Friday possibly to be a fraction of a day or week to move the next or previous midnight on a Friday morning, and then whole additional weeks are added or subtracted to make up # Fridays. 0 goes to the beginning of the day, or the closest Friday before if it is not the right day, independent of sign.

sat add/subtract Saturdays. `"a"` causes the first added or subtracted Saturday possibly to be a fraction of a day or week to move the next or previous midnight on a Saturday morning, and then whole additional weeks are added or subtracted to make up # Saturdays. 0 goes to the beginning of the day, or the closest Saturday before if it is not the right day, independent of sign.

wk add/subtract weeks. `"a"` is not allowed.

tdy add/subtract "ten-day" periods of months (ten-day periods begin on the first, 11th, and 21st of the month but not the 31st). Without `"a"`, the day number of the result is 1, 11, or 21, adding # partial or entire ten-day periods to get there. If `"a"` is used, # must be either 1, 2, or 3, and the time will be midnight. 0 goes to the beginning of the current ten-day period, independent of sign.

mth add/subtract months. `"a"` aligns to nearest # months within the year, starting with January, and # must be a divisor of 12 and less than 12. (For example, 3 aligns to Jan 1, Apr 1, Jul 1, Oct 1 at midnight.) 0 goes to the beginning of the current month, independent of sign.

qtr add/subtract quarters. `"a"` aligns to nearest # quarters within the year, and # must be either 1 or 2. (For example, 2 aligns to Jan 1 or Jul 1 at midnight.) 0 goes to the beginning of the current quarter, independent of sign.

yr add/subtract years. `"a"` aligns to nearest # years. (For example, 5 aligns to Jan 1 at midnight in 1995, 2000, 2005, and so on.) 0 goes to the beginning of the current year, independent of sign.

When relative time objects are added to time/date objects, the fields from an element of the relative time object are parsed and added to the corresponding element of the time/date object from left to right. (If either the time/date object or the relative time object is shorter than the other, it is reused cyclically in the standard S manner.)

For example, for a date/time of May 13, 2012 4:32 PM (a Sunday), the relative time element is: `"+a3hr +12hr -1day"`.

- The first field of the relative time specifies adding up to three hours, and aligning to the nearest three-hour boundary. This operation advances the time to 6:00 PM.
- The second field specifies adding twelve hours, which advances the time to 6:00 AM on May 14.
- The third field specifies subtracting a whole day, which leaves us at 6:00 AM on May 13.

You can add relative time objects to time/date objects, or you can subtract relative time objects from time/date objects. Also, you can add them to each other, subtract them from each other, or multiplt them by integers. When they are negated, the sign of each field is reversed. When they are added together, they are concatenated, so that if `x` is a time/date object and `y` and `z` are relative time objects, `(x + y) + z == x + (y + z)`; however, `y + z` is not the same as `z + y`.

Slots

Data (`character`) a string vector representing the relative time.

holidays (`positionsCalendar`) a vector of holiday dates.

Notes

1. All alignment and other operations are done in the local time zone of the time/date object.
2. The `holidays` slot of the relative time object is used to define which dates besides weekends are not business days; these dates are taken in the `holidays` slots time zone.
3. When adding units of time without the `"a"` flag in the field, the smaller units of time are not changed; for example, when adding days, the time of day stays the same, and when adding months, the day of the month and the time of day stay the same.

See Also

`timeDate` class, `timeSpan` class, `timeRelative` function.

`timeSeq` *Sequences of Times*

Description

Constructs a regularly-spaced `timeDate` object.

Usage

```
timeSeq(from, to, by = "days", length.out, k.by=1, align.by=FALSE,
        extend=FALSE, week.align=NULL, holidays, exceptions,
        additions, format, zone)
```

Arguments

	at least one of `from` or `to` is required, plus `length.out` (the desired length of the resulting sequence). Alternatively can be both `from` and `to`, in which case if `length.out` and `by` are supplied, `length.out` is ignored.
	the starting value of the sequence: a `timeDate` object (or number or character string representing one).
`to`	the ending value of the sequence: a `timeDate` object (or number or character string representing one).
`by`	the spacing between the successive values in the sequence. This can be a `timeSpan`, `timeRelative`, or `numeric` value, in which case `k.by` is ignored. Alternatively, it can be one of the following character strings, giving the time units of intervals between values in the sequence:

```
"milliseconds"
"seconds"
"minutes"
"hours"
"days"
"weekdays"
"bizdays"
"weeks"
"months"
"quarters"
"years"
```

`length.out`	the length of the sequence before additions and exceptions are included.
`k.by`	a non-zero integer giving the width of the interval between consecutive values in the sequence in terms of the units given in `by`. Ignored if `by` is not a character string.
`align.by`	a logical value. If `TRUE`, adjusts the sequence so that each element is on a whole number of the `by` * `k.by` units. For example, if the units are 2 months, the sequence is on only the first of January, March, May, and so on. Ignored if `by` is not a character string.
`extend`	a logical value. If `TRUE` and `align.by` is also `TRUE`, instead of making the entire sequence lie between `from` and `to`, make it extend just past `from` and `to` to the next aligned values. For example, if `from` is January 15th and the sequence is by 1 month units, and if `extend` is `FALSE`, the sequence starts on February 1st, and if it is `TRUE`, January 1st. Ignored if `by` is not a character string.
`week.align`	if `by` is `"weeks"`, you can supply a character string (or a number 0 to 6, with 0 being Sunday) to specify a weekday. The character string must be sufficient to

	make a unique case-insensitive match to the strings in `timeDateOptions("time.day.name")`. The sequence is adjusted so all its elements fall on the given weekday. If `align.by` is `TRUE`, then it is adjusted to start at midnight. The `extend` argument is used to decide which direction to adjust the day. This argument is ignored if `by` is not a character string, or if it is not `"weeks"`.
`holidays`	the holidays for business day sequences (ignored if `by` is not a character string).
`exceptions`	an event object giving any time periods when the sequence should have no values. These are applied after the sequence is created from from/to/by `length.out`.
`additions`	any additional times or dates to put in the sequence.
`format`	the time/date output format for printing.
`zone`	the time zone for the sequence.

Value

returns a time/date object as defined by the arguments.

See Also

`seq`, `timeAlign`, `timeSequence`, `format.timeDate`, `holidays`, `timeEvent`

Examples

```
timeSeq("1/1/1992", "1/10/1992")
timeSeq("1/1/1992", "12/1/1992", by = "months")
timeSeq("1/3/1992", "12/5/1992", by = "months", align.by=TRUE)
timeSeq("1/3/1992", "12/5/1992", by = "months", align.by=TRUE, extend=TRUE)
timeSeq("1/1/1992", "1/31/1992", by = "weeks", align.by=TRUE,
   week.align="Mon")
timeSeq("1/1/1992", "12/31/1992", by="weekdays", exceptions=holidays(1992))
timeSeq("1/1/1992", "1/1/1995", by="months", exceptions=timeEvent("1/1/1993", "12/31/1993"))
## subtract one day from a first-of-month sequence to create
## an end-of-month sequence
timeSeq(from = "2/1/2003", to = "1/1/2004", by = "months" ) - 1
```

---

timeSequence *Create a Time Sequence Object*

Description

Constructs a `timeSequence` object.

Usage

```
timeSequence(from, to, by, length.out, k.by=1, align.by=FALSE,
             extend=FALSE, week.align=NULL, holidays=timeDate(), exceptions,
             additions, format, zone )
```

Arguments

	If any arguments are supplied, exactly three of `from`, `to`, `by`, and `length.out` must be supplied.
	the starting value of the sequence: a `timeDate` object (or number or character string representing one).
`to`	the ending value of the sequence: a `timeDate` object (or number or character string representing one).

`by`	the spacing between successive values in the sequence. This can be a `timeSpan`, `timeRelative`, or `numeric` value, in which case `k.by` is ignored. Alternatively, it can be one of the following character strings giving the time units of intervals between values in the sequence:
	`"milliseconds"`
	`"seconds"`
	`"minutes"`
	`"hours"`
	`"days"`
	`"weekdays"`
	`"bizdays"`
	`"weeks"`
	`"months"`
	`"quarters"`
	`"years"`

`length.out`	the length of the sequence before additions and exceptions.
`k.by`	a non-zero integer giving the width of the interval between consecutive values in the sequence in terms of the units given in `by`. Ignored if `by` is not a character string.
`align.by`	a logical value. If `TRUE`, adjusts the sequence so that each element is on a whole number of the `by` * `k.by` units. For example, if the units are 2 months, the sequence is on only the first of January, March, May, and so on. Ignored if `by` is not a character string.
`extend`	a logical value. If `TRUE` and `align.by` is also `TRUE`, instead of making the entire sequence lie between `from` and `to`, make it extend just past `from` and `to` to the next aligned values. For example, if `from` is January 15th and the sequence is by 1 month units, and if `extend` is `FALSE`, the sequence starts on February 1st If `extend` is `TRUE`, the sequence starts on January 1st. Ignored if `by` is not a character string.
`week.align`	if `by` is `"weeks"`, you can supply a character string (or a number 0 to 6, with 0 being Sunday) to specify a weekday to use. The character string must be sufficient to make a unique case-insensitive match to the strings in `timeDateOptions("time.day.name")`. The sequence is adjusted so all its elements fall on the given weekday. If `align.by` is `TRUE`, then it is also adjusted to start at midnight. In either case, the `extend` argument is used to decide which direction to adjust the day. This argument is ignored if `by` is not a character string, or if it is not `"weeks"`.

holidays	the holidays for business day sequences (ignored if by is not a character string).
exceptions	an event object giving time periods when sequence should not have any values. These are applied after the sequence is created from from/to/by/length.out.
additions	additional times/dates to put in the sequence.
format	the time/date output format for printing.
zone	the time zone for the sequence.

Value

returns a time sequence object as defined by the arguments. Note that the timeSeq function is similar, except that it returns a time/date vector. This function returns a compact time sequence object that retains information about the sequence.

See Also

timeSeq, timeSequence, holidays, timeEvent

Examples

```
timeSequence("1/1/1992", "12/1/1992", by = "months")
timeSequence("1/3/1992", "12/5/1992", by = "months", align.by=TRUE)
timeSequence("1/1/1992", "1/31/1992", by = "weeks", align.by=TRUE,
             week.align="Mon")
timeSequence("1/1/1992", "12/31/1992", by="weekdays", exceptions=holidays(1992))
timeSequence("1/1/1992", "1/1/1995", by="months", exceptions=timeEvent("1/1/1993", "12/31/1993"))
```

---

timeSequence-class *Time Sequence Class*

Description

This class is a compact representation of a time/date vector in an arithemetic sequence.

Details

The timeSequence class extends the positionsCalendar class.

Valid timeSequence objects must contain a single non-NA value in at least three of the from, to, by, and length slots. If all four are present, the length slot is ignored, and a warning message is generated when the sequence is used. If length is present and not being ignored, it must be non-negative (that is, a zero-length sequence is equivalent to timeDate()). Otherwise, to have a valid sequence, adding by to from must go towards to. The default sequence (generated by calling timeSequence() or new("timeSequence")) has length 0.

A timeSequence can be coerced to timeDate using as, and regularly-spaced times/dates (or time/date vectors spaced by regular numbers of months) can be coerced to timeSequence using as. This fails if the input is not a regular sequence within a tolerance given by timeDateOptions("ts.eps").

Most operations that work for timeDate objects also work on timeSequence objects (for example, mathematical functions, arithmetic, comparison operators, and subscripting) by first coercing to a time/date vector. Therefore they do not return timeSequence objects. Because of this, it is more efficient to coerce a timeSequence to timeDate using as before performing an extended set of calculations on the original object, rather than coercing for each operation.

Slots

from (timeDate) the start of the sequence.

to (timeDate) the end of the sequence.

by (timeInterval) the increment for the sequence.

length (integer) the length of the sequence.

exceptions (event) time periods to remove from the sequence.

additions (positionsCalendar) times/dates to add to the sequence.

format (character) the time/date output format for sequence display.

time.zone (character) the time zone for the sequence.

See Also

timeSequence function.

---

timeSpan *Constructor Function For* timeSpan *Class*

Description

Constructs an object of class timeSpan.

Usage

```
timeSpan(charvec, in.format, format, julian, ms)
```

Arguments

charvec	the character vector to parse.
in.format	the time span input format for parsing. Defaults to timeDateOptions("tspan.in.format").
format	the output format to apply to the returned object. Defaults to timeDateOptions("tspan.out.format").
julian	the integer vector of days of the time span. Can be a non-integer if ms is missing, in which case the fractional part represents fractions of days.
ms	an integer vector of milliseconds of the time span.

Details

You can call this function with no arguments. If you supply any arguments, at least one of `charvec`, `julian`, or `ms` must be present.

- If `charvec` is given, the `in.format` is used to parse `charvec` into time spans, and the `julian` and `ms` arguments are ignored (with a warning to the user if they are present).
- If `format` is given, it is put into the `format` slot of the output.
- If `julian` and/or `ms` are provided instead of `charvec`, these values are put into the `timeSpan` as the number of days and milliseconds, respectively, of the time span.

Value

returns a `timeSpan` object constructed from the input. If you provide no arguments, returns the default (empty) `timeSpan` object.

See Also

`timeSpan` class, `format.timeSpan`.

Examples

```
timeSpan()
timeSpan(c( "378d 21h 04min 36s 365MS", "378 d", "1y, 13d, 21h 4MS"))
timeSpan(julian=c(398, 399, 400), ms=c(298392, 3, 0))
```

---

timeSpan-class *Time Span Class*

Description

The `timeSpan` class represents time spans.

Details

The `timeSpan` class is constructed to hold a vector of time spans. It extends the `groupVec` and `groupVecVirtual` classes, as well as `timeInterval`.

The `groupVec` portion of the time class object holds a day portion, stored as an integer vector of the number of full days in each time span, and a time portion, stored as a vector of the number of milliseconds in each time span. The `groupVec` column names are `"julian.day"` and `"milliseconds"`, and the column classes are `integer`. The user can directly change the format specified by the `format` slot (see below), but it is not recommended to change the `groupVec` slots directly.

Slots

columns (`list`) (from `groupVec`).

names (`character`) (from `groupVec`).

classes (`character`) (from `groupVec`).

format (`character`) output format string.

Time span functions

You can create objects of class `timeSpan` by using either the `new` function (in which case they are set up to have length 0 and the default format from `timeDateOptions("tspan.out.format")`), or by using the `timeSpan` function.

`as` relationships are established for `timeSpan` objects to coerce them to and from `character`, `numeric`, and `integer`.

For numbers, the integer part is the number of days, and the fractional part is the fraction of the day given by the number of milliseconds divided by the number of milliseconds in a day. Adding or subtracting numbers to or from `timeSpan` objects works as though the `timeSpan` is converted to a number, the operation is performed, and the number is converted back to a `timeSpan`.

Multiplication and division by numbers are also defined. You can add, subtract, and divide two `timeSpan` objects. (For division, the result is a number.) You can add or subtract a `timeSpan` object to or from a `timeDate` object.

Only a few other mathematical functions make sense for `timeSpan` objects. These are `floor`, `ceiling`, `min`, `max`, `sum`, `mean`, and `range`. Multiplication, division, and operations that do not make sense directly for `timeSpan` objects result in numbers, via automatic coercion to class numeric. In addition to these mathematical operations, all of the standard comparison operators have methods for comparing two `timeSpan` objects.

Input formats

Input format strings are used in the conversion of character strings to `timeSpan` objects. They are read in from left to right, and each format specification encountered is parsed, and the resulting amount of time added to the time span. If the entire input string is not matched by the format string, an NA is inserted into the time span vector. (To skip characters in a string, use %c or %w.)

***** anything not in this list matches itself explicitly.

%c any single character, which is skipped. This can be used with widths and delimiters such as "%3c" (to skip 3 characters) and "%$c" (to skip the rest of the string).

%d input number of days as integer.

%H input number of hours as integer.

%M input number of minutes as integer.

%N input number of milliseconds as integer.

%S input number of seconds as integer.

%w a whitespace-delimited word, which is skipped (no width or delimiter specification. For that, use %c).

%W input number of weeks as integer.

%y input number of 365-day years as integer.

%(digits)(char) For one or more digits between "%" and the specification character, these are parsed as an integer, and specify the field width to be used. The following (digits) characters are scanned for the specified item.

%:(delim)(char) For a colon and any single character between a "%" and the specification character, the field is taken to be all the characters up to but not including the given delimiter character. The delimiter itself is not scanned or skipped by the format.

%$(char) For a $ between a % and a specification character, the field goes to the end of the input string.

whitespace whitespace (spaces, tabs, carriage returns, etc.) is ignored in the input format string. In the string being parsed, any amount of white space can appear between elements of the date/time. Thus, the parse format string " %H:%M: %S " will parse "5: 6:45".

[...] specify optional specification. Text and specifications within the brackets may optionally be included. This does not support fancy backtracking between multiple optional specs.

%%,%[,%] the %, [, and] characters, which must be matched in the input string.

Output formats

Output formats are used to convert `timeSpan` objects to character strings and are stored in the `format` slot of the object. During output, if a given field width is too short to hold the output, the output string becomes "NA". Note that since time spans can be positive or negative, you should use care in specifying field widths. You can use the following format specifications:

***** anything not in this list matches itself explicitly (including whitespace, unlike in input formats).

%d total number of days in span as integer.

%D number of days subtracting off full 365-day years as integer: 1-364.

%E number of days subtracting off 7-day weeks as integer: 1-6.

%H number of hours subtracting off days as integer, 0-23.

%M number of minutes subtracting off hours as integer: 0-59.

%N number of milliseconds in span, subtracting off seconds as integer.

%S number of seconds subtracting off minutes as integer: 0-59.

%s number of seconds subtracting off days as integer.

%W number of 7-day weeks in time span as integer.

%y number of 365-day years in span as integer.

%% the % character.

%(digits)(char) if there are one or more digits between "%" and the specification character, these are parsed as an integer and specify the field width to be used. The value is printed right-justified, using (digits) characters. If (digits) begins with zero, the field is left-padded with zeros if it is a numeric field; otherwise, it is left-padded with spaces. If the value is too long for the field width, the output string becomes "NA" for that time span.

See Also

`groupVec` class, `timeDate` class, `timeDateOptions`, `timeSpan` function, `format.timeSpan`.

Examples

```
## The default format for input is initially:
##   "[%yy[ear[s]][,]] [%dd[ay[s]][,]] [%Hh[our[s]][,]]
##      [%Mm[in[ute][s]][,]] [%Ss[ec[ond][s]][,]] [%NM[s][S]]"
##This allows reading strings such as
##   "378d 21h 04min 36s 365MS", "378 d", "-1y, -13d, -21h -4m"
##The default format for output is initially:
##   "%dd %Hh %Mm %Ss %NMS"
##This results in output such as:
##   "378d 21h 4m 36s 365MS"   "-378d -21h -4m -36s -365MS"
```

---

timeZone-class *Time Zone Classes*

Description

The timeZone classes represent time zones.

Details

The timeZone class is a virtual class for time zones. All time zones classes have an is relationship with timeZone.

The timeZoneC class is a placeholder for a built-in time zone, and it has only one slot, which is the official name of the zone; it extends timeZone.

The timeZoneR class is for user-defined time zones, and also extends timeZone.

'timezone' slots

timeZone is a virtual class and has no slots.

'timezonec' slots

name (character) the name of a built-in time zone.

'timezones' slots

offset (integer) the offset from GMT (in seconds) when not on daylight savings time.

rules (data.frame) rules encoding when to go on daylight savings time (see below).

Built-in zones

The splusTimeDate package contains built-in time zones for the 24 standard time zones around the world. We also include daylight savings time in various areas, and standard time for Central Australia, which is 1/2 hour off Eastern Australia. Currently, the correct daylight savings areas provided are:

- US (1967 and beyond).

- Canada (1974 and beyond).
- New Zealand (1976 and beyond).
- Australia (1973 and beyond).
- Great Britain (1972 and beyond).
- European Union (1977 and beyond).
- Hong Kong (1970 and beyond).

Also, we provide a special time zone for Singapore, which was 7:30 ahead of GMT until May of 1982, when it changed over to 8:00.

The official names of the time zones, in order around the world, are shown below along with their offset from Universal Coordinated Time (UTC, also known as GMT).

st/newzealand Standard time for New Zealand, UTC East 12 hours.

newzealand Standard/summer time for New Zealand.

st/caroline Standard time for Caroline, UTC East 11 hours.

st/eaustralia Standard time for Eastern Australia, UTC East 10 hours.

aust/nsw Standard/summer time for New South Wales, Australia.

aust/tasmania Standard/summer time for Tasmania, Australia.

aust/victoria Standard/summer time for Victoria, Australia.

st/caustralia Standard time for Central Australia, UTC East 9:30 hours.

aust/south Standard/summer time for South Australia.

st/japan Standard time for Japan, UTC East 9 hours.

st/china Standard time for China, UTC East 8 hours.

aust/western Standard/summer time for Western Australia.

hongkong Standard/summer time for Hong Kong.

singapore Standard time for Singapore, reflecting changed zones in 1982.

st/saigon Standard time for Saigon, UTC East 7 hours.

st/kazakh Standard time for Kazakh area, UTC East 6 hours.

st/pakistan Standard time for Pakistan, UTC East 5 hours.

st/caspian Standard time for Caspian Sea area, UTC East 4 hours.

st/moscow Standard time for Moscow, UCT East 3 hours.

st/eeurope Standard time in Eastern European zone, UTC East 2 hours.

europe/east Standard/summer time for EU members, Eastern zone.

st/ceurope Standard time in Central European zone, UTC East 1 hour.

europe/central Standard/summer time for EU members, Central zone.

utc UTC (also known as GMT).

britain Standard/summer time for Great Britain,

europe/west Standard/summer time for EU members, Western zone,

st/azores Standard time for Azores, UTC West 1 hour.

st/oscar Standard time for Oscar, UTC West 2 hours.

st/wgreenland Standard time for Western Greenland, UTC West 3 hours.

can/newfoundland Standard/daylight time for Newfoundland, Canada.

st/atlantic Standard time for Atlantic time zone, UTC West 4 hours.

can/atlantic Standard/daylight Canadian Atlantic time.

st/eastern Standard time for Eastern time zone, UTC West 5 hours.

us/eastern Standard/daylight US Eastern time.

can/eastern Standard/daylight Canadian Eastern time.

st/central Standard time for Central time zone, UTC West 6 hours.

us/central Standard/daylight US Central time.

can/central Standard/daylight Canadian Central time.

st/mountain Standard time for Mountain time zone, UTC West 7 hours.

us/mountain Standard/daylight US Mountain time.

can/mountain Standard/daylight Canadian Mountain time .

st/pacific Standard time for Pacific time zone, UTC West 8 hours.

us/pacific Standard/daylight US Pacific time.

can/pacific Standard/daylight Canadian Pacific time .

st/alaska Standard time for Alaska/Yukon time, UTC West 9 hours.

us/alaska Standard/daylight US Alaska time.

can/yukon Standard/daylight Canadian Yukon time.

st/hawaii Standard time for Hawaii/Alleutian, UTC West 10 hours.

us/hawaii Standard/daylight US Hawaii/Alleutian time.

st/samoa Standard time for Samoa, UTC West 11 hours.

Using zones

You can use the time zones listed in **Built-In Zones** for various operations on `timeDate` objects, including reading times from character data, writing times as character data, and converting between time zones. However, normally the names listed in the table are not used directly, because users want to use the names commonly used in their areas, such as CST for Central Standard Time in the US, Canada, or Australia. The correspondence between "convenient" and "official" names is set up using the `timeZoneList` function.

Defining zones

Besides using the time zones listed in **Built-In Zones**, users can also define their own time zones through use of the `timeZoneR` class. This class allows specification of a time zone with an offset from GMT, in seconds, in the `offset` slot, and a data frame containing rules for when and how to go on daylight savings time in the `rules` slot. (If there is no daylight savings time in this time zone, the data frame should be empty or have 0 rows.)

Each row in the rules data frame encodes a range of years' daylight savings rules; they must be in order to work properly. The rules are encoded in the following columns of the data frame (which must be in order, and all of class integer):

yearfrom	starting year for rules, or -1 to start at the beginning of time.
yearto	ending year for rules, or -1 to end at the end of time.
hasdaylight	indicates whether daylight savings time is used in this year range. Note that this is an integer value, 1 or 0, for
dsextra	offset (in seconds) to add to the regular offset when daylight time is in effect.
monthstart	month (1-12) in which daylight savings time starts (may be before monthend for southern hemisphere).
codestart	code telling how to interpret daystart and xdaystart to calculate the day within the month for starting dayli 1 = start on the daystart (1-31) day of the month. 2 = start on the last daystart (0-Sunday through 6-Saturday) weekday in the month. 3 = start on the first daystart weekday (0-Sunday through 6-Saturday) on or after the xdaystart (1-31) day 4 = start on the last daystart weekday (0-Sunday through 6-Saturday) on or before the xdaystart (1-31) da
daystart	see codestart.
xdaystart	see codestart.
timestart	seconds after midnight local standard time to start daylight savings time, on the day specified by codestart,
monthend	month (1-12) in which daylight savings time ends (may be after monthstart for southern hemisphere).
codeend	code telling how to interpret dayend and xdayend to calculate the day within the month for ending daylight sa 1 = end on the dayend (1-31) day of the month. 2 = end on the last dayend (0-Sunday through 6-Saturday) weekday in the month. 3 = end on the first dayend weekday (0-Sunday through 6-Saturday) on or after the xdayend (1-31) day of the 4 = end on the last dayend weekday (0-Sunday through 6-Saturday) on or before the xdayend (1-31) day of th
dayend	see codeend.
xdayend	see codeend. For examples, see the timeZoneR function. To use user-defined time zones, you must put them i
timeend	seconds after midnight local standard time to end daylight savings time, on the day specified by codeend, etc.

References

Daylight savings boundaries and other time zone information are from the Time Zone Database (often called tz or zoneinfo). See: https://www.iana.org/time-zones.

See Also

timeZoneC function, timeZoneR function, timeZoneList function.

---

timeZoneC *Constructor Function for* timeZoneC *Class*

Description

Constructs a timeZoneC object.

Usage

```
timeZoneC(name)
```

Arguments

name	the official name of a built-in time zone object. Should not be a vector of names. The default is "utc".

Details

The timeZoneC class holds a reference to one of the built-in C time zone objects. See the documentation on the timeZoneC class for more information.

Value

returns a timeZoneC object with the given name, or the default zone name if none is given.

See Also

timeZoneList, timeZoneR, timeZone class.

Examples

```
timeZoneC()
timeZoneC("us/pacific")
```

---

timeZoneConvert *Convert Time Zones*

Description

Convert a timeDate object to a new time zone.

Usage

```
timeZoneConvert(x, zone)
```

Arguments

x	the time vector object to convert.
zone	the time zone to convert to.

Details

Internally, all timeDate objects are stored as times in GMT with an associated time.zone slot. Conversion to the "local" time zone is done only for the purpose of displaying the timeDate object.

Modifying the time.zone slot directly alters the time zone of the object, but not the time itself. (It represents the same instant in a different part of the world.) When displayed, the time component differs according to the time difference betwwen the time zones before and after the change.

The function timeZoneConvert modifies the time zone *and* the actual time by the time difference between the new and old time zones. As a result, the printed display of the timeDate object remains the same (other than any displayed time zone information). This is useful when reading in data from a file without specific time zone information (which, by default, is created with a GMT time zone), and then converting it to a different local time zone without changing the printed appearance of the dates and times.

Value

returns the converted `timeDate` object.

See Also

`timeConvert`, `timeDate`, `timeCalendar`, `holidays`, `timeZoneList`, `timeDate`.

Examples

```
timeDateOptions(time.zone="GMT",
  time.in.format="%m/%d/%Y [%H:%M]",
  time.out.format="%m/%d/%Y %02H:%02M (%Z)")
date1 <- timeDate("3/22/2002 12:00", zone="GMT")
date1
# 3/22/2002 12:00 (GMT)
date2 <- timeZoneConvert(date1, "PST")
date2 # appears the same as date1, except for zone
# 3/22/2002 12:00 (PST)
date1 - date2 # these times are 8 hours apart

# modifying the time.zone slot does not change
# the actual time, just the display
date3 <- date2
date3@time.zone <- "EST"
date3 # displays as 3 hours later
# 3/22/2002 15:00 (EST)
date2-date3 # but the difference is zero
# 0d 0h 0m 0s 0MS
```

---

timeZoneList *Time Zone List*

Description

Returns or modifies the time zone list.

Usage

```
timeZoneList(...)
```

Arguments

...	(see below)

Details

The time zone list is a named list whose names are the character strings that are recognized when you convert strings to time objects, and whose elements are the corresponding time zone objects. (See the documentation for class `timeZone`.) The `timeZoneList` function provides an easy way for the user to define the character strings to use for input of given time zones, and to define user-defined time zones for use in `timeDate` objects. For example, a North American user would probably want to recognize "EST" as the US or Canadian time zone known as Eastern Standard Time, whereas an Australian user might prefer to have "EST" refer to Eastern Australian time. The `timeZoneList` function has the following behavior:

- If no arguments are given, the current time zone list is returned.
- If a single list is given as the argument, its named components are added to the time zone list.
- If multiple named arguments are given, they are added to the list.

In either of the two latter cases, the elements to be added to the list must be time zone objects. The default time zone list has the following built-in zone components. (See documentation on the `timeZone` class for more information.)

Atlantic can/atlantic (Standard/daylight Canadian Atlantic time)

ADT can/atlantic

AST can/atlantic

Halifax can/atlantic

PuertoRico st/atlantic (Atlantic Standard Time, Puerto Rico and Virgin Islands)

Eastern us/eastern (Standard/daylight US Eastern time)

EST us/eastern

EDT us/eastern

EST5EDT us/eastern

EST/EDT us/eastern

Indiana st/eastern (Standard only US/Canadian Eastern time)

Toronto can/eastern (Standard/daylight Canadian Eastern time)

Central us/central (Standard/daylight US Central time)

CST us/central

CDT us/central

CST6CDT us/central

CST/CDT us/central

Chicago us/central

Winnipeg can/central (Standard/daylight Canadian Central time)

Mountain us/mountain (Standard/daylight US Mountain time)

MST us/mountain

MDT us/mountain

MST7MDT us/mountain

MST/MDT us/mountain

Denver us/mountain

Arizona st/mountain (Standard only US/Canadian Mountain time)

Edmonton can/mountain (Standard/daylight Canadian Mountain time)

Pacific us/pacific (Standard/daylight US Pacific time)

PST us/pacific

PDT us/pacific

PST8PDT us/pacific

PST/PDT us/pacific

Vancouver can/pacific (Standard/daylight Canadian Pacific time)

Alaska us/alaska (Standard/daylight US Alaska time)

AKST us/alaska

AKDT us/alaska

AKST/AKDT us/alaska

Aleutian us/hawaii (Standard/daylight US Hawaii/Aleutian time)

HST st/hawaii (Standard only US Hawaii/Aleutian time)

Hawaii st/hawaii

Midway st/samoa (Standard time for Samoa)

Samoa st/samoa

SST st/samoa

Japan st/japan (Standard time for Japan)

Tokyo st/japan

JST st/japan

China st/china (Standard time for China and Western Australia)

HongKong hongkong (Standard/daylight time for Hong Kong)

Singapore singapore (Standard time for Singapore, reflecting changed zones in 1982)

Sydney aust/nsw (Standard/summer time for New South Wales, Australia)

Hobart aust/tasmania (Standard/summer time for Tasmania, Australia)

Melbourne aust/victoria (Standard/summer time for Victoria, Australia)

Adelaide aust/south (Standard/summer time for South Australia)

Darwin st/caustralia (Standard only time for Central Australia)

Perth aust/western (Standard/daylight time for Western Australia)

Auckland newzealand (Standard time for New Zealand)

NZST newzealand

NZDT newzealand

Marshall st/newzealand (Marshall Islands Standard Time)

Wake st/newzealand (Wake Islands Standard Time)

IDLE st/newzealand (International Date Line East)

Chamorro st/eaustralia (Chamorro Standard Time - Guam and Northern Mariana Islands)

ChST st/eaustralia (Chamorro Standard Time - Guam and Northern Mariana Islands)

Yap st/eaustralia (Yap Time)

YAPT st/eaustralia (Yap Time)

Caroline st/caroline (Line Islands Time - Caroline and other Line Islands)

LINT st/caroline (Line Islands Time - Caroline and other Line Islands)

UTC utc (Greenwich Mean Time/Universal Coordinated Time)

GMT utc

GDT britain (Standard time for Great Britain)

London britain (Standard time for Great Britain)

BST britain

WET europe/west (Standard/summer time for EU members, Western zone)

Wes europe/west (Standard/summer time for EU members, Western zone)

WEST europe/west

WET/WEST europe/west

WED europe/west

WEDT europe/west

CET europe/central (Standard/summer time for EU members, Central zone)

CEST europe/central

MET europe/central

MEST europe/central

MET/MEST europe/central

EET europe/east (Standard/summer time for EU members, Eastern zone)

EEST europe/east

EET/EEST europe/east

Value

returns the value of the time zone list before the function call is returned. If arguments are given, it is returned invisibly.

Side Effects

If arguments are given, they are used to modify the current value of `.time.zone.list`, which is assigned in the `splusTimeDate` package environment. It is like `timeDateOptions`, where if you want your entries to the time zone list to persist in subsequent sessions, you should use `timeZoneList` in `.First`.

See Also

`timeZoneC`, `timeZoneR`, `timeZone` class.

Examples

```
# return the entire time zone list
timeZoneList()
# define the string "PDT8PST" to mean US Pacific time
timeZoneList(PDT8PST = timeZoneC("us/pacific"))
# define a time zone for a small island 1/2 hour east of GMT
timeZoneList(small.island = timeZoneR(offset=1800))
```

---

timeZoneR *Constructor Function for* timeZoneR *Class*

Description

Construct a timeZoneR object.

Usage

```
timeZoneR(offset=0, yearfrom=integer(0), yearto=integer(0),
          hasdaylight=logical(0), dsextra=integer(0),
          monthstart=integer(0), codestart=integer(0),
          daystart=integer(0), xdaystart=integer(0),
          timestart=integer(0), monthend=integer(0),
          codeend=integer(0), dayend=integer(0),
          xdayend=integer(0), timeend=integer(0), rules)
```

Arguments

offset	the offset from GMT (in seconds) when not on daylight savings time.
yearfrom	the starting years for rules, or -1 to start at the beginning of time.
yearto	the ending years for rules, or -1 to end at the end of time.
hasdaylight	specifies whether daylight savings time is used in each year range.
dsextra	the offsets (in seconds) to add to the regular offset when daylight time is in effect.
monthstart	the months (1-12) in which daylight savings time starts (can be after monthend for the southern hemisphere).
codestart	the codes telling how to interpret daystart and xdaystart to calculate the days within the month for starting daylight savings time.

1: start on the daystart (1-31) day of the month.
2: start on the last daystart (0-Sunday through 6-Saturday) weekday in the month.
3: start on the first daystart weekday (0-Sunday through 6-Saturday) on or after the xdaystart (1-31) day of the month.
4: start on the last daystart weekday (0-Sunday through 6-Saturday) on or before the xdaystart (1-31) day of the month.

daystart	see codestart.
xdaystart	see codestart.

timestart	the seconds after midnight local standard time to start daylight savings time, on the day specified by codestart and other arguments.
monthend	the months (1-12) in which daylight savings time ends (can be before monthstart for the southern hemisphere).
codeend	the codes specifying interpreting dayend and xdayend to calculate the day within the month for ending daylight savings time.

1: end on the dayend (1-31) day of the month.
2: end on the last dayend (0-Sunday through 6-Saturday) weekday in the month.
3: end on the first dayend weekday (0-Sunday through 6-Saturday) on or after the xdayend (1-31) day of the month.
4: end on the last dayend weekday (0-Sunday through 6-Saturday) on or before the xdayend (1-31) day of the month.

dayend	see codeend.
xdayend	see codeend.
timeend	the seconds after midnight local standard time to end daylight savings time, on the day specified by codeend and other arguments.
rules	a data frame of rules encoding when to go on daylight savings time. (Overrides all other arguments except offset, if it is not missing, and must contain columns corresponding to those arguments in that order, or be completely empty. Note that the columns should all be integer. See timeZoneR class documentation for more information.)

Details

The arguments other than offset and rules define the components of the rules for when to go on daylight savings. Each is a vector with one component for each rule.

The timeZoneR class provides to users a way to define time zones. To use the defined time zones in timeDate objects, they must also be added to the time zone list. (See timeZoneList for more information.)

Value

returns a timeZoneR object with the given name, or the default time zone if no arguments are supplied.

See Also

timeZoneList, timeZoneC, timeZone class.

Examples

```
timeZoneR()
timeZoneR(offset=3*3600)
# time zone with daylight time that changed to daylight time on the
# last Sunday in April and last Sunday in September through 1989,
# and then on the 1st Sunday in May and October thereafter.
# Each time change occurs at 2AM local standard time.
```

```
timeZoneR( offset = 3600,
  yearfrom=c( -1, 1990), yearto=c( 1989, -1 ),
  hasdaylight=c( TRUE, TRUE ), dsextra=c( 3600, 3600 ),
  monthstart=c( 4, 5 ), codestart=c( 2, 3 ),
  daystart=c( 0, 0 ), xdaystart=c(0,1),
  timestart=c( 2*3600, 2*3600 ),
  monthend=c( 9, 10 ), codeend=c( 2, 3 ),
  dayend=c( 0, 0 ), xdayend=c(0,1),
  timeend=c(2*3600, 2*3600))
```

Index