R403: The R Language for Statistical Computing

Topic 11: Working with dates

Kaloyan Ganev

2022/2023

Lecture Contents

Introduction

Introductory notes

- As in regular spreadsheet software, date information should be formatted in a specific way so that it is recognized as such
- R can handle dates and times in several ways
- We will discuss each one in turns
- Why do we need to know this?
- Answer: Time series analysis + the need to use different date and time formats and convert one to another

The Date class

The Date class

- This is R's simplest class for handling dates (but not times)
- Date information is stored in R as integers
- More specifically, "day zero" is Jan 1, 1970
- Every next date corresponds to the count of days from day zero
- Dates before date zero correspond to negative counts
- Uses the Gregorian calendar, so dates earlier than 1752 should be worked with very cautiously

The Date class (2)

- To get the current date, use:

```
Sys.Date()
```

- (The output can be converted to a different format if necessary; more on that later)
- See that date information is formatted as follows: `yyyy-mm-dd`
- So, if you need to store date information in the built-in Date class, just write:

```
datevar1 <- as.Date("2016-10-08")
```

- It is also directly acceptable to use slashes, i.e.: `yyyy/mm/dd`

```
datevar2 <- as.Date("2016/10/08")
```

The Date class (3)

- If you work with data that comes with dates formatted differently, you could use it by specifying a format string
- Format strings are composed of elements shown in the table below

Code	Value
%d	Day of the month (decimal number)
%m	Month (decimal number)
%b	Month (abbreviated) in the current locale
%B	Month (full name) in the current locale
%y	Year (2 digit)
%Y	Year (4 digit)

The Date class (4)

- Examples:

```
Sys.getlocale()
Sys.setlocale(category = "LC_ALL", locale = "English_United
    States.1252")
as.Date("07/31/2016", format = "%m/%d/%Y")
as.Date("December 26, 2011", format = "%B %d, %Y")
as.Date("26MAR2011", format = "%d%b%Y")
```

- If you import date data from MS Excel, note that it uses a different time origin (day zero is Dec 30, 1899)
- You can tell this to R, for example:

```
as.Date(42651, origin = "1899-12-30")
```

The Date class (5)

- To see which number corresponds to a date:

```
as.numeric(datevar1)
```

- You can also calculate differences, e.g. to find out how many days have passed since your birthday:

```
Sys.Date() - as.Date("1991-11-29")
difftime(Sys.Date(), as.Date("1991-11-29"), units = "weeks") #
    could be "auto", "secs", "mins", "hours", "days", "weeks"
```

Time Sequences with the Date Class

- Created using the `seq()` function
  Example: create a sequence of 20 dates starting from Oct 10, 2016:

```
tseq1 <- seq(as.Date("2016-10-10"), by = "days", length = 20)
```

- It is also possible to do the following:

```
tseq2 <- seq(from = as.Date("2016-10-10"), to = as.Date("
    2017-10-10"), by = "3 weeks")
```

The chron package

The **chron** package

- Should be installed additionally
- Allows working with times in addition to dates
- Cannot handle time zones and daylight savings time (yet)
- Simple examples

```
library(chron)
dates1 <- dates(c("03/21/2002", "04/26/12", "01/11/14", "01/28/
    1915", "02/10/2016"))
dates1[5] - dates1[4]
times1 <- times(c("20:05:21", "19:29:59", "11:13:31", "10:29:14",
     "06:48:02"))
datestimes1 <- chron(dates = dates1, times = times1)
```

The **chron** package (2)

- You can also add or subtract numbers

```
datestimes1[2] + 3.5
```

- `range()`, `diff()`, etc. are also possible to use
- As always, further details on additional functions can be found in the documentation

POSIX classes

POSIX classes

- What is POSIX? See `https://en.wikipedia.org/wiki/POSIX`
- In addition to all above, POSIX classes allow handling time zones
- In R there are two standard POSIX date-time classes: `POSIXct` and `POSIXlt`
- `POSIXct` (`ct` stands for "calendar time"): date and time values are stored as the number of seconds since the origin (Jan 1, 1970)
- `POSIXlt` (`ct` stands for "local time"): date and time values are stored as a list; separate elements are used to store seconds, minutes, hours, etc.
- the POSIXct class is usually preferred

POSIX classes (2)

- To get the current date and time in POSIXct form:

```
Sys.time()
```

- Input format:
  - Dates are specified by first writing the year, then the month, and finally the day; the three elements are separated by dashes or slashes:

```
pos1 <- "2016-10-15"
pos2 <- "2016/10/16"
posdate1 <- as.POSIXct(pos1)
posdate2 <- as.POSIXct(pos2)
c(posdate1,posdate2)
```

  - If you need to have also times, they are separated from date by a space and then hours, minutes, and seconds are entered separated by a colon:

```
pos3 <- "2016-10-15 15:45:37"
posdate <- as.POSIXct(pos3)
```

POSIX classes (3)

- An example with using market tick data (high-frequency financial data):

```
tick_data <- read.csv("tick_data.csv", stringsAsFactors = FALSE)
tick_data$DATE <- as.Date(tick_data$DATE, format = "%m/%d/%Y")
tick_data$datetime <- paste(tick_data$DATE, tick_data$TIME, sep =
    " ")
tick_data$datetime <- as.POSIXct(tick_data$datetime)
tick_data <- subset(tick_data, select = c(5,3:4))
```

POSIX classes (4)

- As in the built-in `Date` class, it is possible to specify the format of the date
- Also, there is a command option that allows to explicitly specify the time zone
- Example:

```
date0 <- as.POSIXct("10-10-2016", format = "%d-%m-%Y", tz = "EST"
    )
```

- When you print the result, see that the time zone is displayed as `EEST` (meaning Easter European Summer Time)
- This means that R automatically takes care of daylight saving time; compare with a December date:

```
date0 <- as.POSIXct("10-12-2016", format = "%d-%m-%Y", tz = "EST"
    )
```

- List of time zone abbreviations: https://en.wikipedia.org/wiki/List_of_time_zone_abbreviations

The lubridate package

The **lubridate** package

- Install it so that you can use it
- Provides greater ease of working with `POSIXt`, `Date` and **chron** objects
- Provides compatibility with many time-series objects specific to popular packages such as **zoo**, **xts**, **tSeries**, **fts**, etc.
- Documentation spans lots of pages
- Strings are read into R as POSIXct date-time objects

The **lubridate** package (2)

- Reading is implemented through a series of functions listed in the following table:

Function	Order of elements
`ymd()`	Year, month, day
`ydm()`	Year, day, month
`mdy()`	Month, day, year
`dmy()`	Day, month, year
`hm()`	Hour, minute
`hms()`	Hour, minute, second
`ymd_hms()`[1]	Year, month, day, hour, minute, second

- Example of usage:

```
datetime0 <- dmy_hms("15-09-2016 13:45:01")
```

[1] And permutations of ymd.

The **lubridate** package (3)

- To set the time zone, two options are available
- First, let's get current time; with **lubridate** this is as easy as:

```
nowtime <- now()
```

- If you want to change the way the time instant contained in `nowtime` is displayed (i.e. what was the time of that instant in a different time zone, say UTC[2]):

```
with_tz(nowtime, tzone = "UTC")
```

- If you want to change the actual instant of time contained in `nowtime` but keep the displayed clock time:

```
force_tz(nowtime, tzone = "UTC")
```

[2] Universal Time Zone; same time as GMT.

The **lubridate** package (4)

- To extract the numerical value of the month, day, etc. in nowtime:

```
month(nowtime)
day(nowtime)
hour(nowtime)
```

- To extract respectively the abbreviated and full name:

```
month(nowtime, label = TRUE)
month(nowtime, label = TRUE, abbr = FALSE)
```

- The same stuff can be done for weekdays with the `wday()` command

```
wday(nowtime, label = TRUE)
wday(nowtime, label = TRUE, abbr = FALSE)
```

The **lubridate** package (5)

- Time intervals can also be created, withing the `interval` class

```
yesttime <- nowtime - days(1)
int1 <- interval(nowtime, yesttime)
```

- Two intervals can be checked for overlapping with `int_overlap()` (returns a logical value)
- If they overlap, the overlapping period can be calculated with `setdiff()`

The **lubridate** package (6)

- It is easy to do a lot of arithmetic with **lubridate** objects
- For this purpose, there are the so-called helper functions
- There are two types of helper functions, respectively for creating two classes of time spans: periods and durations
- The functions for creating periods are named after the plurals of time units, e.g.:

```
years(1) + hours(2) + minutes(15) # integer values only allowed
```

- The functions for creating durations just add a “d” in front:

```
dyears(1) # exactly 365 days
```

The **lubridate** package (7)

- Are two classes really needed?
- Check out and compare the results of these:

```
dmy(31012016) + years(1)
dmy(31012016) + dyears(1)
```

- Why we get a difference? 2016 is a leap year!
- Note: **lubridate** is vectorized so it can be applied to vectors too; also it is possible to use it within functions