anytime: Easier Date and Time Conversion

Dirk Eddelbuettel[1]

[1]Department of Statistics, University of Illinois, Urbana-Champaign, IL, USA

This version was compiled on August 10, 2019

The anytime package provides functions which convert from both a number of different input variable types (integer, numeric, character, factor) and different input formats which are tried heuristically offering a powerful and versatile date and time converter that (generally) requires no user input and operates autonomously.

Motivation

R excels at computing with dates, and times. Using a *typed* representation for your data is highly recommended not only because of the functionality offered but also because of the added safety stemming from proper representation.

But there is a small nuisance cost in interactive work as well as in programming. Users must have told `as.POSIXct()` about a million times that the origin is (of course) the epoch. Do we really have to say it a million more times? Similarly, when parsing dates that are *some variant* of the common YYYYMMDD format, do we really have to manually convert from `integer` or `numeric` or `factor` or `ordered` to character? Having one of several common separators and/or date formats (YYYY-MM-DD, YYYY/MM/DD, YYYYMMDD, YYYY-mon-DD and so on, with or without times), do we really need a format string? Or could a smart converter function do this for us?

The `anytime()` function aims to provide such a *general purpose* converter returning a proper `POSIXct` (or `Date`) object no matter the input (provided it was parseable), relying on Boost Date_Time for the (efficient, performant) conversion. `anydate()` is an additional wrapper returning a `Date` object instead. `utctime()` and `utcdate()` are two variants which interpret input as *coordinated universal time* (UTC), *i.e.* free of any timezone.

Examples

We set up the R environment and display for the examples below. Note that the package caches the (local) timezone information (and `anytime:::setTZ()` can be used to reset this value later).

```
Sys.setenv(TZ=anytime:::getTZ()) # TZ helper
library(anytime)                 # caches TZ info
options(width=50,                # column width
        digits.secs=6)           # fractional secs
```

From Integer, Numeric, Factor or Ordered. For numeric dates in the range of the (numeric) *yyyymmdd* format, we use `anydate()`.

```
## integer
anydate(20160101L + 0:2)
#  [1] "2016-01-01" "2016-01-02" "2016-01-03"

## numeric
anydate(20160101 + 0:2)
#  [1] "2016-01-01" "2016-01-02" "2016-01-03"
```

Numeric input also works for datetimes if its range corresponds to the range of `as.numeric()` values of `POSIXct` variables:

```
## integer
anytime(1451628000L + 0:2)
#  [1] "2016-01-01 00:00:00 CST"
#  [2] "2016-01-01 00:00:01 CST"
#  [3] "2016-01-01 00:00:02 CST"

## numeric
anytime(1451628000 + 0:2)
#  [1] "2016-01-01 00:00:00 CST"
#  [2] "2016-01-01 00:00:01 CST"
#  [3] "2016-01-01 00:00:02 CST"
```

This is a change from version 0.3.0; the old behaviour (which was not fully consistent in how it treated numeric input values, but convenient for input in the ranges shown here) can be enabled via either an argument to the function or a global options, see `help(anytime)` for details:

```
## integer
anytime(20160101L + 0:2, oldHeuristic=TRUE)
#  [1] "2016-01-01 CST" "2016-01-02 CST"
#  [3] "2016-01-03 CST"

## numeric
anytime(20160101 + 0:2, oldHeuristic=TRUE)
#  [1] "2016-01-01 CST" "2016-01-02 CST"
#  [3] "2016-01-03 CST"
```

In general, it is now preferred to use `anydate()` on values in this range (or resort to using `oldHeuristics=TRUE` as shown).

Factor or Ordered. Factor variables and their order variant are also supported directly.

```
## factor
anytime(as.factor(20160101 + 0:2))
#  [1] "2016-01-01 CST" "2016-01-02 CST"
#  [3] "2016-01-03 CST"

## ordered
anytime(as.ordered(20160101 + 0:2))
#  [1] "2016-01-01 CST" "2016-01-02 CST"
#  [3] "2016-01-03 CST"
```

Note that `factor` and `ordered` variables may *appear* to be like numeric variables, they are in fact converted to character first and treated just like character input (described in the next section).

Character: Simple. Character input is supported in a variety of formats. We first show simple formats.

```
## Dates: Character
anytime(as.character(20160101 + 0:2))
#  [1] "2016-01-01 CST" "2016-01-02 CST"
#  [3] "2016-01-03 CST"

## Dates: alternate formats
anytime(c("20160101", "2016/01/02", "2016-01-03"))
#  [1] "2016-01-01 CST" "2016-01-02 CST"
#  [3] "2016-01-03 CST"
```

Character: ISO. ISO8661 date(time) formats are supported with both 'T' and a space as separator of date and time.

```
## Datetime: ISO with/without fractional seconds
anytime(c("2016-01-01 10:11:12",
          "2016-01-01T10:11:12.345678"))
#  [1] "2016-01-01 10:11:12.000000 CST"
#  [2] "2016-01-01 10:11:12.345678 CST"
```

Character: Textual month formats. Date formats with month abbreviations are supported in a number of common orderings.

```
## ISO style
anytime(c("2016-Sep-01 10:11:12",
          "Sep/01/2016 10:11:12",
          "Sep-01-2016 10:11:12"))
#  [1] "2016-09-01 10:11:12 CDT"
#  [2] "2016-09-01 10:11:12 CDT"
#  [3] "2016-09-01 10:11:12 CDT"

## Datetime: Mixed format
## (cf http://stackoverflow.com/questions/39259184)
anytime(c("Thu Sep 01 10:11:12 2016",
          "Thu Sep 01 10:11:12.345678 2016"))
#  [1] "2016-09-01 10:11:12.000000 CDT"
#  [2] "2016-09-01 10:11:12.345678 CDT"
```

Character: Dealing with DST. This shows an important aspect. When not working in localtime (by overriding to `UTC`) the *change in difference* to UTC is correctly covered (which the underlying Boost Date_Time library does not do by itself).

```
## Datetime: pre/post DST
anytime(c("2016-01-31 12:13:14",
          "2016-08-31 12:13:14"))
#  [1] "2016-01-31 12:13:14 CST"
#  [2] "2016-08-31 12:13:14 CDT"
## important: catches change
anytime(c("2016-01-31 12:13:14",
          "2016-08-31 12:13:14"), tz="UTC")
#  [1] "2016-01-31 18:13:14 UTC"
#  [2] "2016-08-31 17:13:14 UTC"
```

Technical Details

The actual parsing and conversion is done by two different Boost libraries. First, the top-level R function checks the input argument type and branches on date or datetime types. All other types get handed to a function using Boost lexical_cast to convert from *anything numeric* to a string representation. This textual representation is then parsed by Boost Date_Time to create the corresponding date, or datetime, type. (There are also a number of special cases where numeric values are directly converted; see below for a discussion.) We use the **BH** package (Eddelbuettel *et al.*, 2019a) to access these Boost libraries, and rely on **Rcpp** (Eddelbuettel and François, 2011; Eddelbuettel, 2013; Eddelbuettel *et al.*, 2019b) for a seamless C++ interface to and from R.

The Boost Date_Time library is addressing the need for parsing date and datetimes from text. It permits us to loop over a suitably large number of *candidate formats* with considerable ease. The formats are generally variants of the ISO 8601 date format, *i.e.*, of the YYYY-MM-DD ordering. We also allow for textual representation of months, *e.g.*, 'Jan' for January. This feature is not internationalised.

The list of current formats can be retrieved by the `getFormats()` function. Users can also add to this list at runtime by calling `addFormats()`, as well as removing formats. User-provided formats are tried before the formats supplied by the package.

```
fmts <- getFormats()
length(fmts)
#  [1] 83
head(fmts,10)
#   [1] "%Y-%m-%d %H:%M:%S%f" "%Y-%m-%e %H:%M:%S%f"
#   [3] "%Y-%m-%d %H%M%S%f"   "%Y-%m-%e %H%M%S%f"
#   [5] "%Y/%m/%d %H:%M:%S%f" "%Y/%m/%e %H:%M:%S%f"
#   [7] "%Y%m%d %H%M%S%f"     "%Y%m%d %H:%M:%S%f"
#   [9] "%m/%d/%Y %H:%M:%S%f" "%m/%e/%Y %H:%M:%S%f"
tail(fmts,10)
#   [1] "%d-%b-%Y" "%e-%b-%Y" "%Y-%B-%d" "%Y-%B-%e"
#   [5] "%Y%B%d"   "%Y%B%e"   "%B/%d/%Y" "%B/%e/%Y"
#   [9] "%B-%d-%Y" "%B-%e-%Y"
```

As a fallback for, *e.g.*, different behavior on Windows where Boost does not consult the `TZ` environment variable, and to be generally as close as possible to parsing by the R language and system, we also support the parser from R itself. As R does not expose this part of its API at the C level, we use the **Rcpp** package (Eddelbuettel and François, 2011; Eddelbuettel, 2013; Eddelbuettel *et al.*, 2019b). This code path is enabled when `useR=TRUE` is used.

Output Formats

A related topic is faithful and easy to read *representation* of datetime objects in output, *i.e.,* formatting and printing such objects.

In the spirit of *no configuration* used on the parsing side, formating support is provided via several functions. These all follow different known standards and are accessible by the name of the standard, or, in one case, the non-standard convention. All return a a character representation.

```
pt <- anytime("2016-01-31 12:13:14.123456")
iso8601(pt)
#  [1] "2016-01-31T12:13:14"
rfc2822(pt)
#  [1] "Sun, 31 Jan 2016 12:13:14.123456 -0600"
rfc3339(pt)
#  [1] "2016-01-31T12:13:14.123456-0600"
yyyymmdd(pt)
#  [1] "20160131"
```

Ambiguities

The **anytime** package is designed to operate heuristically on a number of *plausible* and *sane* formats. This cannot possibly cover all conceivable cases.

North America versus the world. In general, **anytime** tries to gently nudge users towards ISO 8601 order of *year* followed by *month* and *day*. But for example in the United States, another prevalent form insists on month-day-year ordering. As many users are likely to encounter such input format, **anytime** accomodates this use provided a separator is used: input with either a slash (/) or a hyphen (-) is accepted and parsed.

Asserts

The **anytime** package also contains two helper functions that can assist in defensive programming by validating input arguments. The `assertTime()` and `assertDate()` functions validate if the given input can be parsed, respectively, as `Datetime` or `Date` objects. In case one of the inputs cannot be parsed, an error is triggered. Otherwise the parsed input is returned invisibly.

Comparison

The **anytime** aims to satisfy two goal: be performant, and the same time flexible in terms of not requiring an explicit input format. We can gauge the relative performance via several pairwise compariosns.

Speed. The `as.POSIXct()` function in R provides a useful baseline as it is also implemented in compiled code. The `fastPOSIct()` function from the **fasttime** package (Urbanek, 2016) excels at converting one (and only one) input format *fast* to a (UTC-only) datetime object. A simple benchmark coverting 100 input strings 100,000 times finds both `as.POSIXct()` and `anytime()` at very comparable and similar performance, but well over one order of magnitude slower that the highly-focussed `fastPOSIXct()`. This result is reasonable: a highly specialised function can (yand should) outperform two (relatively fast) universal converters. `anytime()` is still compelling as it easier to use than `as.POSIXct()` by not requiring a format string (for formats other than ISO 8601).

```
library(rbenchmark)
library(fasttime)
inp <- rep("2019-01-02 03:04:05", 100)
benchmark(fasttime=fastPOSIXct(inp),
          baseR=as.POSIXct(inp),
          anytime=anytime(inp),
          replications=1e5)[,1:4]
#        test replications elapsed relative
#  3  anytime       100000  26.346   20.111
#  2    baseR       100000  24.492   18.696
#  1 fasttime       100000   1.310    1.000
```

Generality. The **parsedate** package (Csárdi and Torvalds, 2019) brings the very general date parsing utility from the git version control software to R. In a similar comparison of 100 input strings parsed 5000 times, we find its `parse_date()` function to be more than an order of magnitude slower than `anytime()` or `as.POSIXct()`. Again, this is reasonable as the greater flexibility of **parsedate** comes at a cost in performance relative to the more restricted alternatives.

```
library(rbenchmark)
library(parsedate)
inp <- rep("2019-01-02 03:04:05", 100)
benchmark(parsedate=parse_date(inp),
          baseR=as.POSIXct(inp),
          anytime=anytime(inp),
          replications=5e3)[,1:4]
#         test replications elapsed relative
#  3   anytime         5000   1.298    1.048
#  2     baseR         5000   1.239    1.000
#  1 parsedate         5000  18.922   15.272
```

All-in. The **lubridate** package (Spinu *et al.*, 2018) is a widely-used package for working with dates and times offering a wide variety of functions. Its parser requires at least a *hint*: the user has to specify whether input is ordered as, say, year-month-day, or day-month-year, or another form. **lubridate** has changed its internals considerably over the years. Early versions did not contain compiled code; a C-based parser was added first, and current versions embed the CCTZ C++ library (White and Miller, 2019) first brought to R by package **RcppCCTZ** (Eddelbuettel, 2019).

While **lubridate** is less general than **anytime** (in that it generally requires user input on the ordering of date elements), it is also slower as can be seen in the example below. The more-widely used form (here `ymd_hms()`) is over an order of magnitude slower; the less well-known function `parse_data_times()` (which still requires *hints*) is still several times slower as shown below.

```
library(rbenchmark)
suppressMessages(library(lubridate))
inp <- rep("2019-01-02 03:04:05", 100)
benchmark(ymd_hms=ymd_hms(inp),
          parse_dt=parse_date_time(inp, "ymd_HMS"),
          anytime=anytime(inp),
          replications=1e4)[, 1:4]
#        test replications elapsed relative
#  3  anytime        10000   2.617    1.000
#  2 parse_dt        10000  21.346    8.157
#  1  ymd_hms        10000  43.639   16.675
```

Summary

We describe the **anytime** package which offers fast, convenient and reliable date and datetime conversion for R users along with helper functions for formatting and assertions. Different types of input are illustrated and described in detail, and performance is analyzed via several benchmark comparisons.

We show that the **anytime** package is no slower than the base R parser, and much faster than either the most flexible parsing alternative, or a commonly-used package in this space—all the while freeing users from having to supply explicit formats specified in advance. The combination of features, performance and ease-of-use may make **anytime** a compelling alternative for R users parsing and analysing dates and times.

References

Csárdi G, Torvalds L (2019). *parsedate: Recognize and Parse Dates in Various Formats, Including All ISO 8601 Formats*. R package version 1.2.0, URL https://CRAN.R-project.org/package=parsedate.

Eddelbuettel D (2013). *Seamless R and C++ Integration with Rcpp*. Springer, New York. doi:10.1007/978-1-4614-6868-4.

Eddelbuettel D (2019). *RcppCCTZ: Rcpp Bindings for the CCTZ LibraryRcpp*. R package version 0.2.6, URL https://CRAN.R-project.org/CRAN=package=RcppCCTZ.

Eddelbuettel D, Emerson JW, Kane MJ (2019a). *BH: Boost C++ Header Files*. R package version 1.69.0-1, URL https://CRAN.R-project.org/package=BH.

Eddelbuettel D, François R (2011). "Rcpp: Seamless R and C++ Integration." *Journal of Statistical Software*, **40**(8), 1–18. doi:10.18637/jss.v040.i08.

Eddelbuettel D, François R, Allaire J, Ushey K, Kou Q, Russel N, Chambers J, Bates D (2019b). *Rcpp: Seamless R and C++ Integration*. R package version 1.0.1, URL https://CRAN.R-project.org/CRAN=package=Rcpp.

Spinu V, Grolemund G, Wickham H, Lyttle I, Constigan I, Law J, Mitarotonda D, Larmarange J, Boiser J, Lee CH (2018). *lubridate: Make Dealing with Dates a Little Easier*. R package version 1.7.4, URL https://CRAN.R-project.org/CRAN=package=lubridate.

Urbanek S (2016). *fasttime: Fast Utility Function for Time Parsing and Conversion*. R package version 1.0.2, URL https://CRAN.R-project.org/package=fasttime.

White B, Miller G (2019). *CCTZ: A C++ library for translating between absolute and civil times using the rules of a time zone*. GitHub Repository, URL https://github.com/google/cctz.