Adding a Date class to S-Plus version 5

Terry M. Therneau

Mayo Foundation

November 5, 1998

1 Introduction

This illustrates the creation of a rather simple class of objects, using the new class and methods code described in Chambers [1]. The class is more complex than many described in that book, however, and might serve as a useful discussion for others (as well as documentation for me!). Thanks is due to Jennifer Hodgdon at StatSci for many useful comments on this document. Any inaccuracies or mistakes belong solely to the primary author, however.

From a purely utilitarian view, the class is not a necesarry addition to Splus, since the language as provided by StatSci contains a new `timeDate` class which is more general. Locally, however, we have a set of functions for an old-style class called "date", which because of familiar useage we would like to retain. (The functions have been part of statlib for several years, slightly predating the `chron` submission, but we are not aware of any wide usage outside of Mayo). The functions could have been left as the old-style classes, of course.

The basic design of a `date` object is constained by the wish for backwards compatability.

- Numerically, the date is an integer, the number of days since January 1, 1960, with 1/1/60 as day 0 and Dec 31 of 1959 as day -1.
- The date contains an optional format which is usually blank. In this latter case a global option controls the printing.
- Print and format methods are calls to as.character, which formats the date accordingly.
- The most used explicit functions were
  - mdy.date: given month, day and year produce a date
  - date.mdy: given a date, return a list containing month, day, year and optionally the weekday
  - as.date: convert character strings such as 3Mar98 into dates, as best as possible.
- There were also subscripting, format, print, numeric, and logical methods defined.

The three basic functions are found in the appendix. Since we're used to calling them, they have been left almost unchanged from the originals posted 5 years ago. The last line of `mdy.date` for instance, changed from

```
class(temp) <- 'date'
tmp
```

to

```
new('date', temp)
```

and this was the only modification.

2 Basic methods

The basic definition of the class is

```
> setClass("date",
       representation("integer", format='character'))

> setAs("numeric", "date",
       function(object) new("date", .Data=floor(object)))

> setAs('character', 'date',
       function(object) as.date(object))
```

We are defining a date as an integer vector with one added attribute, a format. A look at the resultant object tells us that it has two slots:

```
> temp <- mdy.date(m=1:3, d=10:12, y=59:61)
> getSlots(temp)
      .Data      format
  "integer" "character"
```

Next, we define a mapping from numeric objects to dates. The use of the `floor` function is important, since S4 may turn floating point numbers that are not near an integer into NA. The character conversion simply calls the older function. The as.date code knows how to read several formats: '3/12/53', '10Mar98', etc., although it is not as versitile as the new timeDate function.

The printing of dates leaves something to be desired, however:

```
> temp
An object of class "date"

Slot ".Data":
```

```
[1] -356   41  436

Slot "format":
character(0)
```

so our next step is to create adequate show() and print() functions. The common peice underneath all of them is the conversion method from a date to a character string.

```
> setAs('date', 'character', function(object) {
    if (length(object@format))  fmt <- object@format
    else {
        fmt <- .Options[['date.format']]
        if (is.null(fmt)) fmt <- '%d%b%y'
        }
    as.character(timeDate(julian=object@.Data, format=fmt))
    })

> setMethod('show', 'date',
          function(object) {
              str <- as(object, 'character')
              invisible(print.atomic(str,F))
    })

> temp
[1] 10Jan59 11Feb60 12Mar61
```

The first function determines what will occur whenever either `as.character(x)` or `as(x, 'character')` are invoked, and `x` is a date object. The actual work is done by making use of the timeDate functions, after deciding on an appropriate format for the string. The timeDate objects in S-Plus always (almost?) have a format as a part of the object; at the time that a timeDate is created the default format is used to fill in that slot if the user has not specified one. With dates, we choose to use the default system format that is in effect at the time the date is printed, rather than the one in effect when it was created. Thus, date objects usually do not have a format as a part of the object, so the function has to decide on an appropriate default format before calling the timeDate function.

The `show` method is the one called automatically by S for display; is is called with the data as its only argument. For efficiency reasons, new style methods must have an argument list that is exactly identical to the generic. Thus `show` has exactly one argument; no options are allowed. Users, however, may desire a more flexible printing option. We also provide an old-style print method which allows for optional arguments.

```
print.date <- function(x, quote, format, ...) {
    if (missing(format))  str <- as(x, 'character')
    else {
        x@format <- format
        str <- as(x, 'character')
        }
    if (missing(quote)) quote<-F
    invisible(print.atomic(str,quote))
    }

setMethod('format', 'date', function(x) format(as(x, 'character')))
```

This allows the user to override the default format and/or turn on quotes for the string. The ... argument allows other common print options, e.g. `digits`, that are not relevant for date objects to pass through without an error.

The `format` method causes dates to print as desired in several other situations, data.frames in particular. The call to `format` within it ensures that the returned strings are all of the same length. Notice that the `show` method's function has a formal argument of `object`, but the formal argument for the `format` method is `x`. It turns out that the default methods for the former and the latter use `object` and `x`, respectively, and it is critical to match the argument names *exactly*. (The default methods can be viewed by `getMethod('show')` and `getMethod('format')`, respectively.) If the match is not exact a warning message will arise:

```
> setMethod('format', 'date', function(object) as(object, 'character'))
Warning messages:
  the arguments of the method differ from those of the function; a local
        function was created in the method in: setMethod("format", "date",
        function(object) ....
```

In the generic functions discussed below, I found that S does not always detect this problem. Creation of a method with the wrong formal arguments will then result in an 'argument not found' message when the code is invoked.

3 Numeric methods

As currently defined, a date is recognized by S as also being an integer vector. When no specific date method is available, the integer part is extracted and integer methods used:

```
> temp[1:2]
[1] -356   41
> temp+3
[1] -353   44  439
> mean(temp)
[1] 40.33333
> sort(temp)
Problem in sort: Internal error : cannot sort data of mode list: sort(x)
```

This is actually a good default for most mathematical operations. When the date library was first created, I tried to make some of these illegal, i.e., assume that 3*date didn't make any sense. However, exactly this type of thing happens when computing the scale for a plot axis and in several other places; I now believe that the restrictions are counterproductive. The last example shows that method dispatch still isn't perfect; sort is a new style method and should have been called with the integer part of the date.

Certain operations should return a date rather than in integer, however, and we define these.

```
> setMethod('[', 'date', function(x, ..., drop = T) {
        x@.Data <- x@.Data[..., drop=drop]
        x
        })
> setMethod('range', 'date', function(x, ..., na.rm=F) {
        x@.Data <- range(x@.Data, na.rm=na.rm)
        x
        })
> setMethod('max', 'date', function(x, ..., na.rm=F) {
        x@.Data <- max(x@.Data, na.rm=na.rm)
        x
        })
> setMethod('min', 'date', function(x, ..., na.rm=F) {
        x@.Data <- min(x@.Data, na.rm=na.rm)
        x
        })
> setMethod('sort', 'date', function(x, partial=NULL, na.last=NA) {
        x@.Data <- sort(x@.Data, partial=partial, na.last=na.last)
        x
        })
```

```
> setMethod('+', signature(e1='date', e2='numeric'),
          function(e1,e2) {
              e1@.Data <- e1@.Data + round(e2)
              e1
              })

> setMethod('+', signature(e1='numeric', e2='date'),
          function(e1,e2) {
              e2@.Data <- e2@.Data + round(e1)
              e2
              })

> setMethod('-', signature(e1='date', e2='numeric'),
          function(e1,e2) {
              e1@.Data <- e1@.Data - round(e2)
              e1
              })
```

Per section 8.3.2, the `min`, `max` and `range` methods can ignore their ... arguments. Things now work pretty much as we would expect.

```
> #Birthdates of 5 important children
> birth <-  mdy.date(c(9,9,9,4,8), c(29,22,24, 26, 28),
                                c(78, 81, 83, 85,87))
> birth
[1] 29Sep78 22Sep81 24Sep83 26Apr85 28Aug87
> temp2 <- mdy.date(8,22,98)
> birth +2
[1]  1Oct78 24Sep81 26Sep83 28Apr85 30Aug87
> birth-30
```

```
[1] 30Aug78 23Aug81 25Aug83 27Mar85 29Jul87
> 30 - birth
[1]  -6816  -7905  -8637  -9217 -10071
> 2 + birth
[1]  1Oct78 24Sep81 26Sep83 28Apr85 30Aug87

> floor( (temp2-birth)/365.24)  #current ages
[1] 19 16 14 13 10

> temp2 - birth           #age in days
[1] 26Nov79 30Nov76 29Nov74 28Apr73 26Dec70

> range(birth)
[1] 29Sep78 28Aug87
> min(birth)
[1] 29Sep78
> min(birth, temp2)
[1] 6846

> nbirth <- birth
> names(nbirth) <- c("Adam", "Nathan", "Joel", "Isaac", "Elizabeth")
> nbirth
 Adam Nathan Joel Isaac Elizabeth
 6846   7935 8667  9247     10101
> class(nbirth)
[1] "named"
```

Three operations did not come out as we expected. For the subtraction of two dates, S has decided that the closest method match is `date - integer`, since dates are integers after all. It always uses an explicit method rather than a generic one if it thinks the explicit one makes sense. This is easily cured in this instance.

```
> setMethod("-", signature(e1='date', e2='date'),
          function(e1, e2) e1@.Data - e2@.Data)
> temp2 - birth
[1] 7267 6178 5446 4866 4012
```

The second oddity is why `min(birth)` returns a date but `min(birth,birth)` returns an integer. When `min`, `max` or `range` have more than one argument, the generic function concantonates the arguments with the `concat()` function. Methods for concatonation are given using the `concat.two` primitive (if S-Plus knows how to concatonate pairs, it then knows how to concatonate arbitrary lists). Methods are not yet possible for `c()`, however.

```
> setMethod("concat.two", c("date", "date"),
          function(x,y)
              x@.Data <- concat.two(x@.Data, y@.Data)
              x
              )

> min(birth, temp2)
[1] 29Sept78
> concat(birth, temp2)
[1] 29Sep78 22Sep81 24Sep83 26Apr85 28Aug87 22Aug98
> c(birth, temp2)
[1]  6846  7935  8667  9247 10101 14113
```

The issue of named date vectors is not so easily approached. (Translation: it currently has me stumped.)

4 Plotting

The orignal date functions had a plotting method which attempts to make more reasonable axis labels. One deficiency with old style methods is that this special code was only dispatched for the case of a date on the x-axis label, with no provision for y. The new methods allows for more complex 'signatures' in the decision. Here is the basic methods setup.

```
> setMethod('plot', signature('date', 'ANY'),
        function(x,y, ...) plotdate1(x,y,...))
> setMethod('plot', signature("ANY", 'date'),
        function(x,y, ...) plotdate2(x,y,...))
> setMethod('plot', signature('date','date'),
        function(x,y, ...) plotdate3(x,y,...))
```

```
> plotdate1 <- function(x,y, ..., xaxt, xlab, ylab)
    if (missing(xlab)) xlab <- deparse(substitute(x))
    if (missing(ylab)) ylab <- deparse(substitute(y))
    if (missing(xaxt))
        plot(x@.Data, y, ..., xaxt='n',  ...,
                      xlab=xlab, ylab=ylab)
        axis.date(1, x)

    else plot(x@.Data, y, ..., xaxt=xaxt, ...,
                      xlab=xlab, ylab=ylab)
```

```
> plot(birth, sqrt(1:5))
```

The last line of the example above will match the (date, 'ANY') signature, calling the `plotdate1` function. If the user has specificed an `xaxt` argument, then the data and axis are plotted just as they would have been by using the default integer method. (The most common case of this would be `xaxt='n'`, where the user wants to add their own special labels). Otherwise, the routine plots the date as an integer, and then labels the axis using the axis.date routine. This latter routine, found in the appendix, is not particularly bright but is usually better than the raw Julian values.

The `plotdate1` routine itself is not coded into the setMethod call only because it's arguments would not match the signature of the default plot method; we want to explicitly refer to the `xlab` argument. (Section 8.2.3 shows another way to approach this problem using `hasArg` to interrogate the ... list). The `plotdate2` routine (not shown) is a repeat of `plotdate1` but focused on the `yaxt` argument, and `plotdate3` optionally labels both axes.

5 Connection to timeDates

It is tempting to make this class integrate more tightly with the timeDate class found in S-Plus. After all, a date is a timeDate (trivially), and a timeDate contains a date as a portion.

```
> setIs('date', 'timeDate',
         coerce=function(object) {
             if (length(object@format)>0)
                     timeDate(julian=object@.Data, format=object@format)
             else timeDate(julian=object@.Data)
             }
      )

> xx <- as(birth,'timeDate')
> xx
An object of class "timeDate"
format: "%02m/%02d/%Y %02H:%02M:%02S.%03N"
time.zone: "GMT"
[1] "09/28/1978 00:00:00.000" "09/22/1981 00:00:00.000"
[3] "09/24/1983 00:00:00.000" "04/26/1985 00:00:00.000"
[5] "08/28/1987 00:00:00.000"

> birth*3
Problem in birth * 3: Function does not make sense for times

> setIs('timeDate', 'date',
        coerce=function(object)
                new('date', .Data=(object@.Data)[[1]],
                              format=object@format)
        )
Problem in setIs("timeDate", "date", coerce = functi..: can't set the `is'
        relation:no write permission on database splus
```

The first relationship states that a date can always be thought of as a timeDate object, and tells how to do the conversion of internal form automatically. (Because timeDates were not implimented with date, time, and format as separate slots the conversion cannot be completely automatic, using simple inheritance). Unfortunately, this declaration has unintended consequences, as seen in the attempt to multiply `birth` times a number. A date is considered to be "more like" a timeDate than it is like an ANY, and thus we inherit many of the default timeDate methods. The authors of timeDate thought that this multiplication should be illegal, and through them it has become illegal for dates.

```
> xx <- getClass('date', F)
> names(xx@contains)
```

```
[1] "integer"  "timeDate"
```

This states that an explicit integer method, if one were defined, would be closer than the timeDate method (see page 318-319), and of course a date method would be closer yet. Two choices are available, either define explicit methods for all of the cases where we diagree with the timeDate choice, or give up the inclusion. We opt for the latter, and will change the `setIs` call to `setAs`. Thus the package still knows how to convert a date to a timeDate when explicitly asked, but inheritances are not assumed.

The second problem is that we cannot force timeDates to be dates *automatically.* If allowed, this statement might change default inheritance rules for timeDates (just as the reverse did for dates). Because of this, the information must be added to the master dataset for timeDates. That data is part of the S-Plus distribution and the code (quite properly) denies us the right to do so.

After changing both functions to `setAs` form, per below, we get more sensible inheritance, while retaining the knowledge of how to convert.

```
> setAs('date', 'timeDate',
         function(object)
             if (length(object@format)>0)
                     timeDate(julian=object@.Data, format=object@format)
             else timeDate(julian=object@.Data)

      )


> setAs('timeDate', 'date',
        function(object)
                new('date', .Data=(object@.Data)[[1]],
                             format=object@format)
        )
> xx <- as(birth, 'timeDate')
> xx
```

```
An object of class "timeDate"
format: "%02m/%02d/%Y %02H:%02M:%02S.%03N"
time.zone: "GMT"
[1] "09/28/1978 00:00:00.000" "09/22/1981 00:00:00.000"
[3] "09/24/1983 00:00:00.000" "04/26/1985 00:00:00.000"
[5] "08/28/1987 00:00:00.000"

> as(xx, 'date')
Terminating S Session: Signal: bad address signal
```

The last problem above is a bug in the current beta release, I believe. It should be gone by the time you read this.

6 Problems

Some issues that remain

- The bugs claimed above, of course.
- If all of the "set" calls shown above are collected into a single file, `date.s` say, then the command `source('date.s')` will not produce what you expect. There are no error messages, but completely inexplicable behavior. The problem is that some of the functions depend on the results of earlier ones, and the earlier ones are not committed until the source function finishes. Many of them store results into the same object, so there can be a "last save wins" problem as well, similar to multiple people editing the same file. The should be fixed via in later releases via a change to `source`, but a safer method is to use `splus < date.s` at the command line.
- I see no way to remove a setIs or setAs that I later repent of, other than to completely erase the .Data directory and then repopulate it. (Clarification from

a reader: setAs can be removed with `removeMethod("coerce", "date")`, setIs requires removal of the entire class.)

- It is much harder to make an intellegent Makefile. For significant software projects, like survival or rpart, I have always ignored the one that S provides and created a project specific file that knows the full dependency for any given function "zed", i.e., .Data/zed ← zed.s ← SCCS/s.zed.s. Methods don't correspond to files with given names.

- I don't see how to handle name date vectors.

This exercise was all done using Version 5.0 Release 2 Beta 1 on a Sun workstation, and some of the issues above will likely be addressed in subsequent versions.

7 Appendix

The `as.date` function is mostly used to turn character strings into dates, for which it uses a C routine. Although less flexible than the input formats provided with the `timeDate` function, this routine is better at guessing the format de novo. For instance, a command containing mixed formats can be handled.

```
> as.date(c("3/10/53", "March 10,1966", '5-8-1987'))
[1] 10Mar53 10Mar66 8May87

as.date <- function(x, order='mdy', ...)
    if (inherits(x, "date")) x
    else if (is.character(x))
        order.vec <- switch(order,
                                   'ymd'= c(1,2,3),
                                   'ydm'= c(1,3,2),
                                   'mdy'= c(2,3,1),
                              'myd'= c(2,1,3),
                              'dym'= c(3,1,2),
                              'dmy'= c(3,2,1),
                               stop("Invalid value for 'order' option"))
        nn <- length(x)
        temp <- .C("char_date", as.integer(nn),
                              as.integer(order.vec),
                              as.character(x),
                              month=integer(nn),
                              day = integer(nn),
                              year= integer(nn))

        month <- ifelse(temp$month<1 | temp$month>12, NA, temp$month)
        day   <- ifelse(temp$day==0, NA, temp$day)
        year  <- ifelse(temp$year==0, NA, temp$year)

        temp <- mdy.date(month, day, year, ...)

    else if (is.numeric(x)) temp <- new('date', floor(as.vector(x)))
    else stop("Cannot coerce to date format")
    temp
```

If the argument is numeric, then the command treats the data as a string of numbers, each containing the number of days from 1/1/1960. The line above

```
    if (is.numeric(x)) temp <- new('date', floor(as.vector(x)))
```

deserves some comment, why didn't I just use `as(x, 'date')`? Because it doesn't always work. The expression `summary(birth)` produces a vector of 5 numbers, the 0, 25, 50, 75 and 100th percentiles, which are not dates. (Because of interpolation, there is no guarrantee that the values would even be integers.) Nevertheless, we might occassionally want to view them as dates.

```
> birth
 [1] 28Sep78 22Sep81 24Sep83 26Apr85 28Aug87

> summary(birth)
  Min. 1st Qu. Median  Mean 3rd Qu.  Max.
  6845  7935    8667   8559  9247   10100
> xx <- summary(birth)
> is.numeric(xx)
[1] T
> as(xx, 'date')
Problem: class "date" has 2 slots, got 4 elements in list
> class(xx)
[1] "table"
```

Although `xx` is a numeric, it is not a simple vector of numbers; `floor(xx)` is also of class 'table' is not a valid entry for the .Data slot of a date. However, `as.vector(x)` is valid, since it removes the extra class information. This modification should perhaps be made to the `setAs` code instead.

The `mdy.date` and `date.mdy` functions are based on formulas in Numerical Recipies.

```
#
#  Return the month, day, and year given a julian date
#
date.mdy <- function(date, weekday=F)
    sdate <- as(date, 'integer')
    sdate <- sdate + 2436935  #From SAS to Num Recipies base point
    wday <- as.integer((sdate+1)%%7 +1)
    temp <- ((sdate-1867216) -.25) / 36524.25
    sdate <- ifelse(sdate >=2299161, trunc(sdate+1+temp -trunc(.25*temp)),
                                        sdate)
    jb <- sdate + 1524
    jc <- trunc(6680 + ((jb-2439870)-122.1)/365.25)
    jd <- trunc(365.25 * jc)
    je <- trunc((jb-jd)/ 30.6001)
    day <- (jb - jd) - trunc(30.6001*je)
    month <- as.integer(ifelse(je>13, je-13, je-1))
    year  <- as.integer(ifelse(month>2, jc-4716, jc-4715))
    year  <- as.integer(ifelse(year <=0, year-1, year))
    if (weekday) list(month=month, day=day, year=year, weekday=wday)
    else         list(month=month, day=day, year=year)



#
#  Get the Julian date, but centered a la SAS, i.e., Jan 1 1960 is day 0.
#    Algorithm taken from Numerical Recipies.
#
mdy.date <- function (month, day, year, nineteen=T, fillday=F, fillmonth=F)
    temp <- any( (month != trunc(month)) | (day != trunc(day)) |
                 (year != trunc(year)))
    if (!is.na(temp) && temp)
        warning("Non integer input values were truncated in mdy.date")
        month <- trunc(month)
        day <- trunc(day)
        year <- trunc(year)

    if (nineteen)  year <- ifelse(year <100, year+1900, year)
```

```
# Force input vectors to be the same length, but in a way that gives an
#   error if their lengths aren't multiples of each other.
temp <- 0*(month + day + year)
month <- month + temp
day   <- day + temp
year  <- year + temp

if (fillmonth)
    temp <- is.na(month)
    month[temp] <- 7
    day[temp] <- 1

if (fillday) day[is.na(day)] <- 15



month[month<1 | month>12] <- NA
day[day<1] <- NA
year[year==0] <- NA     #there is no year 0
year <- ifelse(year<0, year+1, year)
tyear<- ifelse(month>2, year, year-1)
tmon <- ifelse(month>2, month+1, month+13)

julian <- trunc(365.25*tyear) + trunc(30.6001*tmon) + day - 715940
# Check for Gregorian calendar changeover on Oct 15, 1582
temp <- trunc(0.01 * tyear)
save <- ifelse(julian>=-137774, julian +2 + trunc(.25*temp) - temp, julian)

#check for invalid days (31 Feb, etc.) by calculating the Julian date of
#    the first of the next month
year <- ifelse(month==12, year+1, year)
```

```
        month<- ifelse(month==12, 1, month+1)
        day <- 1
        tyear<- ifelse(month>2, year, year-1)
        tmon <- ifelse(month>2, month+1, month+13)
        julian <- trunc(365.25*tyear) + trunc(30.6001*tmon) + day - 715940
        temp <- trunc(0.01 * tyear)
        save2<- ifelse(julian>=-137774, julian +2 + trunc(.25*temp) - temp, julian)
        temp <-as.integer(ifelse(save2>save, save, NA))

        new("date", temp)
```

The last function is an axis routine for dates. If the range of dates is small it uses what the `pretty` function suggests, but if it spans multiple years it labels January 1 of each year.

```
    axis.date <- function(side=1, x)
        x <- x[!is.na(x)]

        xd<- date.mdy(x)
        temp <- pretty(x@.Data, 5)
        delta <- temp[2] - temp[1]
        if (delta <1)
                temp <- seq(min(x), max(x), 1)
        else if (delta > 182)    #try to do it in years
            temp <- xd$year + (x - mdy.date(1,1,xd$year))/365
            temp <- pretty(temp,5)
            temp <- mdy.date(1, 1, floor(temp)) + floor((temp%%1)*365)
            temp <- temp@.Data
```

```
xlim <- par("usr")[2*side - 1:0]
temp <- temp[temp>xlim[1] & temp<xlim[2]]
if (side==1) axis(side, temp, format(as(temp, 'date')))
else  # work around a bug: srt/crt aren't forgotton
    axis(side, temp, format(as(temp, 'date')), srt=90, crt=90)
    axis(side, temp[1], "", srt=0, crt=0)
```