SIOB 296 Introduction to Programming with R

Eric Archer (eric.archer@noaa.gov)

Week 8 (May 23, 2017) - Time/Date, Packages, Files/Folders

Reading

The Book of R: Appendix A - Installing R and Contributed Packages

The Art of R:
Appendix B - Installing and Using Packages

---

do.call (applying elements of a list to a function)

There are times when we have multiple elements in a list that we would like to use as arguments in a function. A common example is having a list of equal length vectors that we want to combine together as rows of a matrix. For example, if we have this list:

```
x <- lapply(1:5, function(i) runif(10, 0, 10))
x
```

```
[[1]]
 [1] 6.1361830 3.1759754 2.7467789 7.7422840 0.2305182 7.2196582 1.6596092
 [8] 8.1650029 8.3917009 4.5865222

[[2]]
 [1] 8.1545974 1.4571507 8.6523474 8.6862861 0.3938074 6.7550540 8.6519322
 [8] 0.6642054 2.1223694 1.4861006

[[3]]
 [1] 2.6103109 3.5015404 1.6034100 4.8245555 4.5005743 0.5318321 4.6849775
 [8] 6.5308194 3.1854246 9.1733973

[[4]]
 [1] 9.9755782 7.2884181 0.8621828 2.8351761 0.2860779 0.7756054 2.1195531
 [8] 4.1879581 6.4680213 6.6049174

[[5]]
 [1] 9.181273 9.485194 8.477084 1.742282 7.795598 2.194463 4.740858
 [8] 2.224811 0.487236 8.277479
```

We could use `rbind()` to make the matrix:

```
rbind(x[[1]], x[[2]], x[[3]], x[[4]], x[[5]])
```

```
         [,1]     [,2]      [,3]     [,4]      [,5]      [,6]     [,7]
[1,] 6.136183 3.175975 2.7467789 7.742284 0.2305182 7.2196582 1.659609
[2,] 8.154597 1.457151 8.6523474 8.686286 0.3938074 6.7550540 8.651932
[3,] 2.610311 3.501540 1.6034100 4.824555 4.5005743 0.5318321 4.684978
[4,] 9.975578 7.288418 0.8621828 2.835176 0.2860779 0.7756054 2.119553
```

```
[5,] 9.181273 9.485194 8.4770839 1.742282 7.7955982 2.1944625 4.740858
          [,8]     [,9]    [,10]
[1,] 8.1650029 8.391701 4.586522
[2,] 0.6642054 2.122369 1.486101
[3,] 6.5308194 3.185425 9.173397
[4,] 4.1879581 6.468021 6.604917
[5,] 2.2248106 0.487236 8.277479
```

But if the list was long, we wouldn't want to type out every element. In order to do this for a list of any length, we can use the `do.call()` function. `do.call()` takes as its first argument, the name of the function we want to use, in this case, `rbind`. The second argument is a list where the elements will be used as arguments to the specified function:

```
do.call(rbind, x)
```

```
         [,1]     [,2]      [,3]     [,4]      [,5]      [,6]     [,7]
[1,] 6.136183 3.175975 2.7467789 7.742284 0.2305182 7.2196582 1.659609
[2,] 8.154597 1.457151 8.6523474 8.686286 0.3938074 6.7550540 8.651932
[3,] 2.610311 3.501540 1.6034100 4.824555 4.5005743 0.5318321 4.684978
[4,] 9.975578 7.288418 0.8621828 2.835176 0.2860779 0.7756054 2.119553
[5,] 9.181273 9.485194 8.4770839 1.742282 7.7955982 2.1944625 4.740858
          [,8]     [,9]    [,10]
[1,] 8.1650029 8.391701 4.586522
[2,] 0.6642054 2.122369 1.486101
[3,] 6.5308194 3.185425 9.173397
[4,] 4.1879581 6.468021 6.604917
[5,] 2.2248106 0.487236 8.277479
```

If the list elements are named, the names are treated as argument names in the function call:

```
sample.args <- list(replace = TRUE, x = letters[1:5], size = 20)
do.call(sample, sample.args)
```

```
 [1] "e" "c" "a" "c" "e" "b" "a" "c" "d" "a" "e" "a" "d" "d" "a" "b" "c"
[18] "a" "c" "e"
```

`do.call()` can be very handy when combined with `lapply()`, `tapply()`, or `by()`, if the result of every iteration is a data.frame and the desire is to combine them all into one data.frame. For example, we want calculate the mean temperature, salinity, and pH for every depth value at each CTD station:

```
ctd <- read.csv("ctd.csv", stringsAsFactors = F)

# a function to summarize the values
smrzStation <- function(df) {
  smry <- data.frame(
    temp = tapply(df$temp, df$depth, mean, na.rm = TRUE),
    salinity = tapply(df$salinity, df$depth, mean, na.rm = TRUE),
    ph = tapply(df$ph, df$depth, mean, na.rm = TRUE)
  )
  # add depth column
  smry$depth <- sort(unique(df$depth))
  smry
}

# summarize each station
station.smry <- by(ctd, ctd$station, smrzStation)
```

```r
# add station name to each data.frame
station.smry <- lapply(names(station.smry), function(x) {
  df <- station.smry[[x]]
  df$station <- x
  df
})

# combine all data.frames
station.smry <- do.call(rbind, station.smry)

head(station.smry)
```

```
      temp salinity       ph depth   station
1 17.22627 33.48636 8.187241     1 Station.1
2 17.18102 33.48907 8.186552     2 Station.1
3 17.07373 33.48578 8.186724     3 Station.1
4 16.92864 33.48302 8.183276     4 Station.1
5 16.75797 33.47880 8.180517     5 Station.1
6 16.54356 33.47014 8.176897     6 Station.1
```

```r
tail(station.smry)
```

```
         temp salinity       ph depth   station
2420 12.79559 33.47910 8.008448    24 Station.9
2520 12.73102 33.48156 8.001552    25 Station.9
2619 12.68508 33.48481 7.992931    26 Station.9
2718 12.64000 33.48937 7.985690    27 Station.9
2817 12.62220 33.49064 7.979138    28 Station.9
2914 12.77365 33.48267 7.990784    29 Station.9
```

Here's another example using `strsplit()` on the `sample_date` column to create a matrix of year, month, and day:

```r
date.split <- strsplit(ctd$sample_date, "-")
head(date.split)
```

```
[[1]]
[1] "2012" "11"   "08"

[[2]]
[1] "2012" "04"   "19"

[[3]]
[1] "2010" "01"   "06"

[[4]]
[1] "2014" "02"   "06"

[[5]]
[1] "2011" "01"   "05"

[[6]]
[1] "2015" "02"   "03"
```

```r
date.mat <- do.call(rbind, date.split)
colnames(date.mat) <- c("year", "month", "day")
```

```
head(date.mat)
```

```
     year   month day
[1,] "2012" "11"  "08"
[2,] "2012" "04"  "19"
[3,] "2010" "01"  "06"
[4,] "2014" "02"  "06"
[5,] "2011" "01"  "05"
[6,] "2015" "02"  "03"
```

Date/Time

In base R, dates and times are primarily stored in one of two related formats, `POSIXlt` and `POSIXct`. "POSIX" is the acronym for "Portable Operating System Interface for uniX", a set of computing standards for maintaining compatability between operating systems. `POSIXlt` stores dates in a list, while `POSIXct` stores dates as character strings. Internally, both store dates as the number of seconds since January 1, 1970. `POSIXct` is the form you are likely to use most and is suitable for storing in data frames. With `POSIXlt`, you can easily extract components:

```
# POSIXlt stores dates in a list
dt.lt <- as.POSIXlt("2011/08/23 6:05")
str(dt.lt)
```

```
 POSIXlt[1:1], format: "2011-08-23 06:05:00"
```

```
dt.lt$mon
```

```
[1] 7
```

```
# years since 1900
dt.lt$year
```

```
[1] 111
```

```
# POSIXct stores dates as character
dt.ct <- as.POSIXct("2011/08/23 6:05")
str(dt.ct)
```

```
 POSIXct[1:1], format: "2011-08-23 06:05:00"
```

```
# this will throw an error because dt.ct is not a list
dt.ct$mon
```

```
Error in dt.ct$mon: $ operator is invalid for atomic vectors
```

character to date (`strptime()` and `as.POSIXlt()`)

There are a few standard date and time character formats that `as.POSIXct` and `as.POSIXlt` can recognized. See the Details section in `?as.POSIXct` for more information. If you want to specify the format, use `strptime()` which takes a character vector of date/time representations, and a character string specifying the format using `%` notation. See the Details section in `?strptime` for a complete list. You can also use `as.POSIXlt()` and specify the `format` argument. Both functions return an object of `POSIXlt`, so to add it to a data.frame, we have to convert it to `POSIXct`.

```
dt <- strptime(ctd$sample_date, format = "%Y-%m-%d")
str(dt)
```

```
 POSIXlt[1:77641], format: "2012-11-08" "2012-04-19" "2010-01-06" "2014-02-06" "2011-01-05" ...
```

```
ctd$date <- as.POSIXct(dt)
str(ctd)
```

```
'data.frame':   77641 obs. of  10 variables:
 $ station    : chr  "Station.1" "Station.1" "Station.1" "Station.1" ...
 $ sample_date: chr  "2012-11-08" "2012-04-19" "2010-01-06" "2014-02-06" ...
 $ temp       : num  16.8 10.5 15.1 14 14.2 ...
 $ salinity   : num  33.4 33.8 33.4 33.4 33.3 ...
 $ dox        : num  8.07 3.16 7.22 7.31 7.91 6.45 3.32 6.14 8.82 6.98 ...
 $ ph         : num  8.2 7.73 8.13 NA 8.16 8.05 7.75 7.94 8.22 NA ...
 $ pct_light  : num  90.3 88.1 89 88 86.2 ...
 $ density    : num  24.3 25.9 24.7 25 24.8 ...
 $ depth      : int  16 18 32 41 3 51 16 48 7 45 ...
 $ date       : POSIXct, format: "2012-11-08" "2012-04-19" ...
```

numeric to date

If you want to specify dates as some number of seconds since a specific date, you can also use `as.POSIXlt()` and specify the `origin` argument. For example, here's 10,000 seconds from May 23, 2017:

```
as.POSIXlt(10000, origin = "2017-05-23")
```

```
[1] "2017-05-22 19:46:40 PDT"
```

Note that it seems to be backwards in time. However, it is because `origin` is assumed to be in GMT, but by default, the time is printed in the local time zone. To see the output in GMT, we have to specify that as the `tz` argument:

```
as.POSIXlt(10000, tz = "GMT", origin = "2017-05-23")
```

```
[1] "2017-05-23 02:46:40 GMT"
```

If you have vectors of numbers representing components of dates and times, those can be converted to `POSIXct` using `ISOdatetime()`, or if you have just date compoents, use `ISOdate()`:

```
dt.mat <- do.call(rbind, strsplit(ctd$sample_date, "-"))
head(dt.mat)
```

```
     [,1]   [,2] [,3]
[1,] "2012" "11" "08"
[2,] "2012" "04" "19"
[3,] "2010" "01" "06"
[4,] "2014" "02" "06"
[5,] "2011" "01" "05"
[6,] "2015" "02" "03"
```

```
dt.iso <- ISOdate(dt.mat[, 1], dt.mat[, 2], dt.mat[, 3])
head(dt.iso)
```

```
[1] "2012-11-08 12:00:00 GMT" "2012-04-19 12:00:00 GMT"
[3] "2010-01-06 12:00:00 GMT" "2014-02-06 12:00:00 GMT"
[5] "2011-01-05 12:00:00 GMT" "2015-02-03 12:00:00 GMT"
```

date to character

To convert a `POSIXlt` or `POSIXct` to character, use the `format()` or `strftime()` functions. Both take the `format` argument to specify how you want the dates and times formatted.

```
# todays date and time in POSIXct format:
now <- Sys.time()
str(now)
```

```
 POSIXct[1:1], format: "2017-05-21 17:34:20"
```

```
now.ch <- strftime(now, format = "%Y-%m-%d %H hours, %M minutes")
now.ch
```

```
[1] "2017-05-21 17 hours, 34 minutes"
```

```
# with both strftime and format, you can specify the output time zone
format(now, "Year: %Y, Month: %m, Day: %d at %H%M", tz = "EST")
```

```
[1] "Year: 2017, Month: 05, Day: 21 at 1934"
```

elapsed time

Times can be subtracted to get elapsed time, the result of which is a `difftime` object.

```
# how long until Christmas?
xmas <- as.POSIXct("2017-12-25")
wait.time <- xmas - Sys.time()
str(wait.time)
```

```
Class 'difftime'  atomic [1:1] 217
  ..- attr(*, "units")= chr "days"
```

`difftime` object have the numeric value of the time difference and an attribute called `units` that specifies in what units the time difference is given. The units of a `difftime` object can be extracted with `units()`:

```
units(wait.time)
```

```
[1] "days"
```

It can also be changed (with an appropriate conversion of the value), by assignment:

```
units(wait.time) <- "secs"
wait.time
```

```
Time difference of 18775540 secs
```

Simple subtraction will automatically choose the units based on the magnitude of the difference. If we want to specify the units, we can use the `difftime()` function:

```
difftime(xmas, Sys.time(), units = "weeks")
```

```
Time difference of 31.04421 weeks
```

`difftime` objects can also be added to dates:

```
# 2 weeks from now:
Sys.time() + as.difftime(2, units = "weeks")
```

```
[1] "2017-06-04 17:34:20 PDT"
```

math on dates

Several simple mathematical summaries can be done on a range of dates:

```
# average date
mean(ctd$date)
```

```
[1] "2013-01-23 16:18:30 PST"
```

```
# range of dates:
range(ctd$date)
```

```
[1] "2010-01-05 PST" "2016-12-20 PST"
```

extract components of dates

There are several functions for extracting components of date objects:

```
weekdays(Sys.time())
```

```
[1] "Sunday"
```

```
months(Sys.time())
```

```
[1] "May"
```

```
quarters(Sys.time())
```

```
[1] "Q2"
```

```
# number of days since the beginning of the year:
julian(Sys.time(), "2017-1-1")
```

```
Time difference of 140.6905 days
```

Finally, a handy built-in vector for labelling months is `month.abb`:

```
month.abb
```

```
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov"
[12] "Dec"
```

This can be useful for adding a factor column in our data.frame:

```
# extract numeric column
i <- as.numeric(substr(ctd$sample_date, 6, 7))
# translate to month abbreviation vector
month <- month.abb[i]
# create a factor with levels in the proper order:
month.fac <- factor(month, levels = month.abb)
table(month.fac)
```

```
month.fac
 Jan  Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct  Nov  Dec
5535 8374 5580 4672 8391 5552 5667 8413 5664 5731 8406 5656
```

Most of what you will need to do with dates can be done with the base R functions. However, there are other packages that some processes more convenient that are worth exploring if you will be working with dates frequently in a specific way: `chron`, `date`, `zoo`, `lubridate`.

Packages

Packages are sets of related functions along with their documentation that can be loaded so they are available to you for use. Prior to loading the package, it has to be installed. Most packages are installed from the central R repository, CRAN. This downloads the package to your system and installs it in your "library". The function for this is `install.packages()`. The first argument is the package name. If you don't have a default CRAN mirror repository set, you'll need to supply the URL to the `repos` argument. A complete list of mirrors can be found at https://cran.r-project.org/mirrors.html.

```
install.packages("swfscMisc", repos = "http://cran.stat.ucla.edu")
```

```
Installing package into '/Users/eric.archer/Library/R/3.4/library'
(as 'lib' is unspecified)
```

```
The downloaded binary packages are in
    /var/folders/pc/g_3vty9n52nfs3bc58fn3t4018168m/T//RtmpB75Ywa/downloaded_packages
```

Using RStudio, packages can also be installed with the "Install" menu option under the "Packages" tab. Once you have the package installed, in order to use its functions, you have to load it with the `library()` function.

```
# The isBetween function doesn't exist yet because swfscMisc hasn't been loaded
exists("isBetween")
```

```
[1] FALSE
```

```
library(swfscMisc)
```

```
Loading required package: mapdata
```

```
Loading required package: maps
```

```
# Now it does...
exists("isBetween")
```

```
[1] TRUE
```

To see a quick description of the package and its functions, use `library(help = pkg)`:

```
library(help = swfscMisc)
```

To get an index of help files for functions in a package, use `help(package = pkg)`:

```
help(package = swfscMisc)
```

To remove a package (but still keep it installed), use `detach(package:pkg)`:

```
detach(package:swfscMisc)
exists("isBetween")
```

```
[1] FALSE
```

Sometimes in a function, you want to load a package and test if it exists. If it doesn't you'll want to raise an error because the rest of the function cannot properly execute without access to that packages functions. This is done with two functions. The first is `require()` which loads the package much like `library()`, but returns `TRUE` if the package can be loaded, and `FALSE` if it cannot. The second function, `stopifnot()` throws an error if its argument is `FALSE`. They are commonly used together as `stopifnot(require(pkg))`.

```
# try to load swfscMisc
stopifnot(require(swfscMisc))
```

```
Loading required package: swfscMisc
```

```
exists("isBetween")
```

```
[1] TRUE
```

```
# try to load a made-up package that I don't have
stopifnot(require(wmwmw))
```

```
Loading required package: wmwmw
```

```
Warning in library(package, lib.loc = lib.loc, character.only = TRUE,
logical.return = TRUE, : there is no package called 'wmwmw'
```

```
Error: require(wmwmw) is not TRUE
```

File/Folder Management

list files in a folder

The `dir()` function lists all of the files in a folder. Keep in mind that it returns a character vector that can be saved to an object to be used later. The `pattern` argument, lists only files that match the specified pattern. Setting `full.names = TRUE` will return the full path of the files. Setting `recursive = TRUE` will provide a list to all subdirectories.

```
# here's a full listing of all .rdata files in the parent folder
files <- dir("..", pattern = ".rdata", full.names = TRUE, recursive = TRUE)
head(files)
```

```
[1] "../Prep/merge data.rdata"
[2] "../Prep/test ws.rdata"
[3] "../Prep/xy.rdata"
[4] "../Week 03 - April 18/three things.rdata"
[5] "../Week 03 - April 18/xy.rdata"
[6] "../Week 04 - April 25/merge data.rdata"
```

We can test if a file or folder is present with `file.exists()`:

```
dir()
```

```
 [1] "ctd.csv"              "format ctd.R"         "free text.txt"
 [4] "lm.R"                 "merge data.rdata"     "multiYearCTD.csv"
 [7] "regression example.R" "species.csv"          "tblCodeSpecies.csv"
[10] "test ws.rdata"        "test.csv"             "trawl.csv"
[13] "Week 1.Rmd"           "Week 2.Rmd"           "Week 3.Rmd"
[16] "Week 4.Rmd"           "Week 5.Rmd"           "Week 6.Rmd"
[19] "Week 7.Rmd"           "Week 8.Rmd"           "Week_3.pdf"
[22] "Week_4.pdf"           "Week_5.pdf"           "Week_6.pdf"
[25] "Week_7.pdf"           "Week_8.pdf"           "Week_8.Rmd"
[28] "x.r"                  "xy.rdata"
```

```
file.exists("missing.rdata")
```

```
[1] FALSE
```

```
x <- 1
save(x, file = "x test.rdata")
file.exists("x test.rdata")
```

```
[1] TRUE
```

To delete a file, use `file.remove()`:

```
file.remove("x test.rdata")
```

```
[1] TRUE
```

```
file.exists("x test.rdata")
```

```
[1] FALSE
```

We can create a new directory with `dir.create()`:

```
dir.create("new dir")
dir()
```

```
 [1] "ctd.csv"              "format ctd.R"         "free text.txt"
 [4] "lm.R"                 "merge data.rdata"     "multiYearCTD.csv"
 [7] "new dir"              "regression example.R" "species.csv"
[10] "tblCodeSpecies.csv"   "test ws.rdata"        "test.csv"
[13] "trawl.csv"            "Week 1.Rmd"           "Week 2.Rmd"
[16] "Week 3.Rmd"           "Week 4.Rmd"           "Week 5.Rmd"
[19] "Week 6.Rmd"           "Week 7.Rmd"           "Week 8.Rmd"
[22] "Week_3.pdf"           "Week_4.pdf"           "Week_5.pdf"
[25] "Week_6.pdf"           "Week_7.pdf"           "Week_8.pdf"
[28] "Week_8.Rmd"           "x.r"                  "xy.rdata"
```

In order to create paths to files that are correct regardless of the OS you're using, use the `file.path()` function:

```
x <- 1
x.fname <- file.path("new dir", "x ws.rdata")
x.fname
```

```
[1] "new dir/x ws.rdata"
```

```
save(x, file = x.fname)
dir("new dir", full.names = TRUE)
```

```
[1] "new dir/x ws.rdata"
```

To remove all path specifications of a filename, use `basename()`:

```
rdata.files <- dir("..", pattern = ".rdata", recursive = TRUE)
head(rdata.files)
```

```
[1] "Prep/merge data.rdata"
[2] "Prep/new dir/x ws.rdata"
[3] "Prep/test ws.rdata"
[4] "Prep/xy.rdata"
[5] "Week 03 - April 18/three things.rdata"
[6] "Week 03 - April 18/xy.rdata"
```

```
head(basename(rdata.files))
```

```
[1] "merge data.rdata"   "x ws.rdata"         "test ws.rdata"
[4] "xy.rdata"           "three things.rdata" "xy.rdata"
```

The reverse, `dirname()` returns just the path portion:

```
dirname(rdata.files)
```

```
[1] "Prep"                "Prep/new dir"        "Prep"
[4] "Prep"                "Week 03 - April 18" "Week 03 - April 18"
[7] "Week 04 - April 25" "Week 05 - May 02"
```

Finding files of a particular extension requires the use of regular expressions. The regular expression that we need is "\.ext$", which matches all strings with ".ext" at the end. So, to find all ".csv" files, we use:

```
csv.fnames <- dir("..", pattern = "\\.csv$", full.names = TRUE, recursive = TRUE)
head(csv.fnames, 5)
```

```
[1] "../Prep/ctd.csv"             "../Prep/multiYearCTD.csv"
[3] "../Prep/species.csv"         "../Prep/tblCodeSpecies.csv"
[5] "../Prep/test.csv"
```

To remove the extension, we use the same string with the `gsub()` function:

```
csv.fnames <- gsub("\\.ext$", "", csv.fnames)
head(basename(csv.fnames), 5)
```

```
[1] "ctd.csv"             "multiYearCTD.csv"   "species.csv"
[4] "tblCodeSpecies.csv" "test.csv"
```

You can't delete a directory that is not empty with `file.remove()`. For this, you need to use `unlink()`. You should include the `recursive = TRUE` argument to delete all files and subdirectories contained in the directory being deleted:

```
unlink("new dir", recursive = TRUE)
dir.exists("new dir")
```

```
[1] FALSE
```